EP0717353

Publication Title:

Efficient and secure update of software and data

Abstract:

Abstract of < 12e0 strong>EP0717353

The invention concerns apparatus for updating a data file, from an earlier version to a later version. The invention compares the earlier version with a later version, and derives a transformation, which contains information as to the similarities and differences. The invention then processes the earlier version, using the transformation, in order to derive the later version, without reference to the later version itself. The invention allows multiple versions of an original file, such as bank records located at multiple locations, to be updated by transmitting the transformation to the multiple locations, instead of transmitting the updated files themselves. The procedure is highly resistant to interception of the updated data, because the transformation, in general, does not contain the entire contents of the later versions.

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 717 353 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
19.06.1996 Bulletin 1996/25

(51) Int. Cl.^{6}: **G06F 9/44**

(21) Application number: **95308717.8**

(22) Date of filing: **01.12.1995**

(84) Designated Contracting States:
DE FR GB

(30) Priority: **14.12.1994 US 355889**

(71) Applicant: **AT&T Corp.**
New York, NY 10013-2412 (US)

(72) Inventors:
- **Korn, David Gerard**
New York, New York 10003 (US)
- **Vo, Kiem-Phong**
Berkeley Heights, New Jersey 07922 (US)

(74) Representative: **Watts, Christopher Malcolm Kelway, Dr. et al**
AT&T (UK) Ltd.
5, Mornington Road
Woodford Green Essex, IG8 0TU (GB)

(54) Efficient and secure update of software and data

(57) The invention concerns apparatus for updating a data file, from an earlier version to a later version. The invention compares the earlier version with a later version, and derives a transformation, which contains information as to the similarities and differences. The invention then processes the earlier version, using the transformation, in order to derive the later version, without reference to the later version itself.

The invention allows multiple versions of an original file, such as bank records located at multiple locations, to be updated by transmitting the transformation to the multiple locations, instead of transmitting the updated files themselves. The procedure is highly resistant to interception of the updated data, because the transformation, in general, does not contain the entire contents of the later versions.

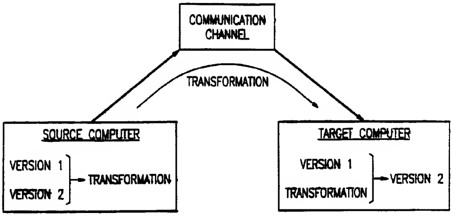


FIG. 1

Printed by Rank Xerox (UK) Business Services
2.12.4/3.4

Description

An APPENDIX is attached which contains source code used by the invention. The invention concerns a method for remotely updating software or data, or both, which reduces the amount of data transmitted, and is less susceptible to interception.

Background of The Invention

Electronic data transmission between computers via a common-carrier telephone system requires speed and security. Transmission speed can be enhanced by compressing the data before transmission, while security can be protected using encryption. Data compression is also beneficial for encryption because it reduces redundancy in the data source, and increases resistance to cracking attempts.

However, if a data file is compressed and encrypted, the resulting compressed-and-encrypted data is based on a single source. Data compression and encryption based on single data sources are not perfect. Unless great redundancy exists in the data, compression does not perform well, and high-speed computers working together with modern crypto-analytic methods can crack most encrypted data.

In many situations, multiple later versions of an initial data source are frequently transmitted after slight changes in the data. The fact that the changes are slight means that all versions are inherently similar. The inherent similarity can be exploited to compute a minimal transformation that transforms one version into another.

This process of computing such a transformation is called data differencing. Data differencing enhances transmission speed because it reduces the amount of data to be communicated. It also enhances privacy because a snooper without the earlier version cannot deduce too much information from the transmitted data.

A number of compression and differencing techniques have been previously considered. The article "A Universal Algorithm for Sequential Data Compression," by J. Ziv and A. Lempel, contained in IEEE Transactions on Information Theory, 23(3): pp. 337 - 343, May, 1977, describes a technique to compress a single data file. This works by analyzing a data sequence and, at each position, if possible, identifying the longest segment that matches another segment in the already analyzed part. Such a segment, if found, is encoded using the matching position and the length of the match. Unmatched data are output as-is.

An implementation of this algorithm was the subject of U.S. Patent number 4,464,650, August 7, 1984, by W.L. Eastman, A. Lempel, and J. Ziv. The Eastman-Lempel-Ziv compression method is slow on de-compression because it has to do roughly the same amount of work during de-compression as during compression. It also does not apply to data differencing.

On UNIX systems, differences between text (not binary) files can be computed using the program "diff," which produces a set of lines that must be deleted or added to transform the first file to the second. The "diff" approach can produce very large transformations because small changes in text lines cause them to be considered different. Further, it only works on text files, so it has a limited range of applicability.

The article "The String-to-String Correction Problem with Block Moves," by Walter F. Tichy, contained in ACM Transactions on Computer Systems, 2(4): pp. 309 - 321, November, 1984, describes an algorithm to compute a set of instructions called block-moves to transform one version of a data file to another. A block-move is a data segment in the second version that matches another segment in the first version. This algorithm is essentially unrealized and does not take advantage of any redundancy inherent in the second version to minimize the transformation needed to construct the second version from the first version.

Summary of The Invention

In one form of the invention, a computer program compares a first and a second, later, version of a file, and produces a sequence of instructions. The instructions allow replication of the second version, based on the first version.

The instructions are of two types: One type is a COPY instruction, which orders copying of a specified sequence of characters into a file being constructed. The specified sequence can reside either in the first version, or in the file under construction.

The second type is an ADD instruction, which orders addition of a sequence of characters, which accompanies the ADD instruction. When the instructions are executed in sequence, they produce a concatenation of sequences, which are COPIED and ADDED, producing a replica of the second version.

Brief Description of The Drawings

Figure 1 is a schematic of how the invention is used for data communication between computers.

Figures 1A through 1I explain, by analogy, principles used by the invention.

Figure 1J illustrates a security aspect of the invention.

Figure 2 gives an example of two versions of a data set and a set of instructions to transform one of the versions to the other.

Figures 2A through 2E illustrate operation of the procedure described in Figure 3.

Figures 2F through 2H illustrate operation of the procedure described in Figure 5.

Figure 3 describes the decoding procedure in a pseudo-C language.

Figure 4 shows the actual encoding of the instructions of FIgure 2.

Figure 5 describes the encoding procedure.

Figure 6 describes the INSERT procedure used in Figure 5.

Figure 7 describes the SEARCH procedure used in Figure 5.

Figure 8 describes the EXTEND procedure used in Figure 5.

Figure 9 shows the inserted positions of the example VERSION1 and VERSION2 data of Figure 2.

Figure 10 shows the secure version of the coded instructions of Figure 4.

Detailed Description of The Invention

A simplified analogy will explain some of the more fundamental aspects of the invention. A technical explanation of computer code which implements part of the invention is given later, beginning in the section entitled "The Invention".

Analogy

Assume that a newspaper reporter and a newspaper editor are writing an article together, but at different locations. Assume that they both have written VERSION 1, which is shown in Figure 1A. Assume that the reporter has made the changes shown in Figure 1B, resulting in VERSION 2, which is shown in Figure 1C (the boxes indicate the locations of changes), and wishes to send this version to the editor. The following procedure will allow the reporter to send VERSION 2 to the editor, but without sending the entire contents of VERSION 2.

Procedure

First, the reporter assigns a position to each word in VERSION 1, as indicated in Figure 1A. There are 50 positions, corresponding to the 50 words.

Second, the reporter ascertains the vocabulary of VERSION 1. The vocabulary is a list, or table, of the words used, as indicated in Figure 1A. Note that each word appears in the vocabulary only once. For example, the word "them" at position 18 is not entered into the vocabulary, because this word previously appeared, at position 16.

In order to assure that each word appears only once, the reporter, as each word is entered, checks to see whether that word already exists in the table. For example, the reporter enters the first word, "when." Then, the reporter takes the word at position 2, "buying," and checks it against the first word in the vocabulary, namely, "when". There is no match, so the reporter enters "buying" into position 2 in the vocabulary.

The reporter proceeds in this manner until the word "them" is reached at position 18. When checking this word against the previous 17 entries in the vocabulary, the reporter finds that "them" matches "them" at position 16 in the vocabulary. Consequently, the latter "them" is not entered into the vocabulary, because it would duplicate the previous "them."

When the last word in VERSION 1 is checked for matches, the vocabulary becomes complete. VERSION 1, which is 50 words in overall length, contains only 38 different words, as indicated by the length of the vocabulary.

Next, the reporter establishes a vocabulary for VERSION 2, as shown in Figure 1C. VERSION 2 has the same vocabulary as VERSION 1, with the exception of three additional words, indicated as NEW WORDS. Now the reporter is ready to transmit VERSION 2 to the editor.

The reporter transmits VERSION 2 by transmitting the following six messages, or instructions, each of which will be explained by a respective Figure in Figures 1D - 1I:

1. COPY 2 1
2. ADD 1 Hawaiian
3. COPY 3 3
4. ADD 1 all
5. COPY 37 8
6. ADD 2 a brick.

Instruction 1

By pre-arrangement, the reporter and the editor set a POINTER at position 1, as indicated in Figure 1D. The first instruction, "COPY 2 1" means "COPY the sequence of 2 words which begin at position 1 in VERSION 1." (In each instruction, it is implicit that the words are placed at the POINTER in VERSION 2.) This operation is shown in Figure 1D.

Note that the syntax is

COPY [number of words, starting position in VERSION 1].

After following the instruction, the editor moves the POINTER by the "number of words" in the COPY instruction, which, in this case, is 2. The POINTER is now located as shown in Figure 1E.

Instruction 2

Instruction 2 states "ADD 1 Hawaiian," which means "ADD the one word 'Hawaiian'." This operation is shown in Figure 1E. Note that the syntax is

ADD [number of words, words themselves].

After following the instruction, the editor moves the POINTER by the "number of words" in the ADD instruction, which, in this case, is 1. The POINTER is now located as shown in Figure 1F.

Instruction 3

The third instruction, "COPY 3 3" means "COPY the sequence of 3 words which begin at position 3 in VERSION 1." This operation is shown in Figure 1F. Note that the syntax is identical to that of instruction 1.

After following the instruction, the editor moves the POINTER by the "number of words" in the copy instruction, which, in this case, is 3. The POINTER is now located as shown in Figure 1G.

Instruction 4

Instruction 4 states "ADD 1 all," which means "ADD the one word 'all'." This operation is shown in Figure 1G.

After following the instruction, the editor moves the POINTER by the "number of words" in the ADD instruction, which, in this case, is 1. The POINTER is now located as shown in Figure 1H.

Instruction 5

The fifth instruction, "COPY 37 8" means "COPY the sequence of 37 words which begin at position 8 in VERSION 1." This operation is shown in Figure 1H. Note that the syntax is identical to that of instruction 1.

After following the instruction, the editor moves the POINTER by the "number of words" in the copy instruction, which, in this case, is 37. The POINTER is now located as shown in Figure 1I.

Instruction 6

Instruction 4 states "ADD 2 a brick," which means "ADD the 2 words 'a brick'." This operation is shown in Figure 1I. Now VERSION 2 has been constructed.

Significant Features

VERSION 2, shown in Figure 1C, contained 43 words, which themselves contained 163 characters. However, the set of six instructions contained 27 characters, plus the instructions themselves (ie, COPY and ADD). If each instruction and each character is encoded as one byte, then the total message equals 27 + 6 characters, or 33 characters. Transmitting 33 characters takes a significantly shorter time than transmitting 163 characters.

The Invention

The previous analogy was a simplification, and does not illustrate all of the features of the invention. Figure 2 illustrates two VERSIONs of a file, together with a TRANSFORMATION, which allows construction of VERSION2 from the combination of VERSION1 and the TRANSFORMATION itself. The procedure shown in Figure 3 accomplishes the reconstruction, and will now be explained. First, the general pattern will be elaborated.

As background, four important points should be recognized. One, each character in each VERSION occupies a numbered position, as shown in Figure 2A. For example, in VERSION 1, the first "a" occupies position 0. The first "b" occupies position 1, and so on.

Two, the numbered positions in VERSION 2 begin after the highest-numbered position in VERSION 1. Thus, since the highest position in VERSION 1 is "15," VERSION 2 begins with "16."

Three, a pointer (which is a variable in the procedure of Figure 3) indicates a current position, as indicated in Figure 2A.

Four, the TRANSFORMATION contains instructions. There are two types of instruction, namely, ADD and COPY. The operation of these two instructions will be explained by showing how VERSION 2 is constructed from VERSION 1.

Instruction 1

In Figure 2B, instruction 1 of the TRANSFORMATION is executed. This instruction, "COPY 4 0," states, in effect, "<u>COPY</u> the <u>FOUR</u> characters beginning at position <u>ZERO</u> in VERSION 1 to the pointer in VERSION 2." (The underlined words refer to the words in the instruction: <u>FOUR</u> refers to 4, and so on.) Figure 2B illustrates the operation.

It is significant that,' in this instruction, the characters used to generate VERSION 2 are obtained from VERSION 1, not from the instruction itself. Thus, if the instruction were obtained via a telephone transmission, an eavesdropper would obtain no information usable in constructing VERSION 2 because the eavesdropper is presumed to have no access to VERSION 1.

The syntax is

COPY [number of characters, starting position]

Instruction 2

In Figure 2C, instruction 2 is executed. This instruction, "ADD 2 x,y," states, in effect, "<u>ADD</u> the <u>TWO</u> characters <u>x</u> and <u>y</u> to VERSION 2, beginning at the pointer."

It is significant that, in this instruction, the characters used to generate VERSION 2 are obtained from the instruction itself. Consequently, an eavesdropper does obtain some information about VERSION 2. However, in practice, this type of instruction is expected to be intermixed with so many types of COPY instructions (which contain no information) that the eavesdropper would obtain no significant information about VERSION 2.

Nevertheless, it is possible, in theory, that the TRANSFORMATION may contain ADD instructions exclusively, in which case the eavesdropper would obtain VERSION 2 in its entirety. Or the number of ADD instructions may be very large. To prevent an eavesdropper from obtaining information from these ADD instructions, an encryption option is provided, which is discussed later.

The syntax is of the ADD instruction is

ADD [number of characters, characters themselves]

Instruction 3

In Figure 2D, instruction 3 is executed. This instruction, "COPY 6 20", in effect, states:

generate a <u>COPY</u>, which is <u>six</u> characters long, using the characters residing between position <u>TWENTY</u> and the pointer, and place them at the pointer.

The result is the sequence xyxyxy, as indicated.

(As another example, if the COPY statement read "COPY 6 19", then it, in effect, would have stated

generate a <u>COPY</u>, which is <u>six</u> characters long, using the characters residing between position <u>NINETEEN</u> (not TWENTY) and the pointer, and place them at the pointer.

The sequence added at the pointer would have been dxydxy, because the sequence residing between position nineteen and the pointer is dxy.)

It is significant that, in this instruction, the characters used to generate VERSION 2 are obtained from VERSION 2 itself. Thus, it is now seen that the COPY instruction uses two sources, depending on the address specified. In this instruction, the address (ie, the "20" in "COPY 6 20") indicates VERSION 2 (ie, the address is greater than 15). VERSION 2 is the source for this command.

In contrast, if the address were 15 or less, then VERSION 1 would have been used as the source.

This approach, in effect, allows one to expand the vocabulary, using ADD instructions. Then, when VERSION 2 is found to contain characters which are (a) not in VERSION 1, but (b) contained in the expanded vocabulary (because added earlier), a COPY instruction can be used. The COPY instruction has two benefits.

One is that a COPY instruction can insert a large number of characters, merely by indicating their total number, and starting position. In contrast, the ADD instruction requires the characters themselves to be transmitted, which entails a longer transmission.

A second is that, as discussed above, the COPY instruction contains no information, unless an interceptor of the instruction also happens to possess VERSION 1.

Instruction 4

In Figure 2E, instruction 4 is executed. This instruction, "COPY 5 9," in effect, states: "COPY the FIVE characters beginning at position NINE in VERSION 1." This instruction resembles instruction 1.

Reference to Process of Figure 3

Figure 3 illustrates a process by which a computer can construct VERSION 2 from the combination of VERSION 1 and the TRANSFORMATION.

In Figure 3, line 2 initializes a variable c, which is used to compute the pointer shown in Figures 2B - 2E. The pointer is computed in lines 9, 13, and 14, as appropriate.

Instruction 1

Instruction 1 (COPY 4 0) of the TRANSFORMATION shown in Figure 2 is executed in line 13 in Figure 3. The variable "p" indicates the starting position of the source data, is obtained from the instruction, and equals 0. Since "p" is less than "n" (line 12), the source lies in VERSION 1. Consequently, line 13 is executed, because of the IF-statement, and the data is copied from VERSION 1, beginning at position VERSION1+p, which is position 0. The variable "s" indicates the number of characters, is obtained from the instruction in line 7, and equals 4.

The pointer is updated on line 16.

Instruction 2

Instruction 2 (ADD 2 x,y) is executed on line 9, because of the IF-statement on line 8. It places a sequence of characters, which is of length "s" (which equals 2) at the location VERSION2 + c. The pointer is updated on line 16.

Instruction 3

Instruction 3 (COPY 6 20) is executed on line 14, because of line 12: "p" (obtained from the instruction, and which equals 20) is less than "n." The copy function uses VERSION2 as the source, beginning at VERSION2 + c. The copy function takes a sequence which is s characters long. (The variable "s" is obtained from pointer-minus-20. The "20" is obtained from the instruction.)

Instruction 4

Instruction 4 is executed on line 13, like instruction 1.

Generation of TRANSFORMATION

Figure 5 contains a procedure which examines each VERSION, character-by-character, and generates the set of instructions which allows construction of VERSION 2 from the combination of VERSION 1 and the instructions, which are called a transformation. The transformation of Figure 2 was discussed above. The general approach taken by this procedure is the following.

Run 1

VERSION 1 is processed first. VERSION 1 is shown at several locations in Figure 2F. The code of Figure 5 asks whether the four-character sequence beginning at position 0 matches a four character sequence beginning at a previous position. Of course, since nothing exists prior to position 0, the answer is No. Thus, position 0 is flagged with a carat, as indicated in the row labeled RUN 1.

Run 2

On the second run, the code asks a similar question, namely, whether the four-character sequence beginning at position 1 matches a four-character sequence beginning at a previous position. The answer is No, and a flag is placed at position 1.

Runs 3 and 4

In a similar way, flags are placed at positions 2 and 3, resulting in the flagging shown at RUN 4 in Figure 2F.

Run 5

Run 5 produces different results. In RUN 5, the code asks the usual question, Does the four-character sequence beginning at position 4 match a four-character sequence beginning at a previous position ? The answer is Yes, at position 0, as indicated in the row labeled RUN 5. Position 4 is not flagged, and the EXTEND function (line 16, Figure 5) is called into action.

The EXTEND function asks how long the match extends. It asks whether the next position after the matched block (the matched block occupies positions 4 - 7, so the next is position 8) has a match located four positions earlier. The answer is Yes, as indicated in trial TA.

It then asks whether the second position after the matched block, position 9, has a match located four positions earlier. The answer is again Yes, as indicated by trial TB. This inquiry continues until a position is reached, in trial TE, where no such match is found.

The EXTEND function has thus ascertained that the match extends for eight positions, from position 4 through position 11. Because of the logic of the code, the last three positions of the extended match are flagged, namely, the "BCD" located in "RUN 5 RESULT."

Run 6

Run 6 asks whether the four-character sequence beginning at position 12 (ie, the rightmost "e") matches any four-character sequence beginning at a position occurring previously. The answer is No, and a flag is placed at position 12, as indicated in the row labeled RUN 6.

Results

The results are shown at the bottom of Figure 2F, and labeled RESULT. Two important features are the following:

One, as will be later explained, the flags indicate not only the beginnings, but also the endings, of possible sequences which are used to construct VERSION 2.

Two, there are regions which are not flagged, as indicated. A search for sequence to use in constructing VERSION 2 never begins in these regions, although the search can begin outside these regions, and then invade the region. Thus, since initial search points are eliminated, the overall searching time is reduced.

Processing of VERSION 2

First Four Runs

VERSION 2, the updated version, is processed next. The code asks whether the four-character sequences beginning at positions 16, 17, 18, and 19, shown in Figure 2G, have matches which begin at previous positions. The four-character sequence beginning at position 16 matches with that beginning at 0, but the latter three do not, so positions 17, 18, and 19 are flagged, as indicated at the top of Figure 2G.

The EXTEND function finds no extended match. The code of Figure 5 jumps to line 29, where it issues the "COPY 4 0" instruction, which was explained above, and appears in Figure 2.

The overall results are indicated by the arrow labeled OUTPUT. Three characters have been flagged, and a COPY instruction has been issued. It is important to note that these flagged characters are later treated as search initialization points, just as are the flagged characters in VERSION 1.

Run 5

The code of Figure 5 asks whether the four-character sequence "xyxy", beginning at position 20 (which resides in VERSION 2, shown in Figure 2G, in the section labeled RUN 5), has a match which begins at a previous position. The code searches, starting at each flag, including those now existing in VERSION 2.

Further, at each flag, the code searches both forward and backward. (This forward-backward searching was not mentioned previously, for simplicity.) The overall searching is indicated by RUN 5 in Figure 2G. Because the searching (a) starts with each flag and (b) searches for four characters, both forward and backward, every four-character sequence of trial T1 through T12 is examined. In addition, similar searching is done, starting with the flagged positions in VERSION 2, namely, positions 17, 18, 19, and 20.

No match is found, so the "x" at position 20 is flagged.

Run 6

The code of Figure 5 asks whether the four-character sequence "yxyx", beginning at position 21 (which resides in VERSION 2), has a match which begins at a previous position. As for position 20, the code searches, forward and backward, beginning at each flag, including those in VERSION 2. No match is found, so position 21 is flagged. The current flagging is indicated at the upper right part of Figure 2H.

Run 7

The code of Figure 5 asks whether the four-character sequence "xyxy", beginning at position 22 (which resides in VERSION 2), has a match which begins at a previous position. The answer is Yes, at position 20. Thus, position 22 is not flagged, as indicated in Figure 2H, and the EXTEND function (line 16 of Figure 5) is entered.

The EXTEND function determines, in the manner described in connection with Figure 2F, that the match extends through position 27 in Figure 2H. Consequently, the last three positions of the extended match are flagged, resulting in the flagging shown in the box at the lower left of the Figure.

At this time, the logic proceeds to line 29 of Figure 5, wherein the two instructions "ADD 2 x,y" and "COPY 6 20" are issued. (The arguments "add" and "c" in line 29 refer to the ADD instruction. The arguments "pos" and "len" refer to the COPY instruction.)

Run 8

The code asks whether the sequence "bcde", beginning at position 28 (See OUTPUT box in Figure 2I), has a match beginning at a previous position. The answer is Yes, at position 9 in VERSION 1 (see Figure 2G, upper left). Now the EXTEND function is called, which determines that the match extends to an additional character, the "f".

The code, at line 29, then issues the instruction "COPY 5 9". The four instructions of Figure 2 have been generated. VERSION 2 can now be reconstructed from VERSION 1, plus the instructions.

Characterization

The preceding overview illustrates the following principles:

ONE, once it is determined that a position represents a previously matched sequence, that position is not flagged, and consequently, subsequent searches are not initiated at that position, because such initiation would be redundant. Thus, a reduced set of searching positions is obtained (if possible).

TWO, VERSION 2 is constructed from two types of sources, namely, (a) sequences copied from either VERSION 1 or VERSION 2 and (b) sequences ADDED to VERSION 2. Further, the ADDED sequences can be used later, in copying operations.

From another point of view, if a first user possesses VERSION 1, and if a second user possesses both VERSION 1 and VERSION 2, the first user can construct a replica of VERSION 2, if the second user identifies the following sequences of characters:

(a) those to be copied from the first user's VERSION 1, (b) those to be added to first user's VERSION 1, and
(c) those to be copied from the first user's replica of VERSION 2.

THREE, each flagged position represents the starting point of a possible vocabulary word. Analogous words are shown in Figure 1A. However, unlike the vocabulary of Figure 1A, the vocabulary of each flagged position represents numerous possible sequences.

For example, assume VERSION 1 is the following:

a b c d e F g h i j k.

If the upper-case F (position 6) is flagged, then it can represent the following sequences:

f fg fgh fghi fghij fghijk.

These sequences can contain other flagged positions.

Therefore, a single flagged position can represent multiple sequences for copying into VERSION 2. Thus, this flagged position can appear in multiple COPY instructions. For instance, "COPY 3 6" means COPY "fgh." "COPY 5" means COPY "fghij".

Security Aspects

As mentioned above, the ADD instructions contain information as to the content of VERSION 2, and this information may need security protection. One approach to security is shown in Figure 1J.

The ADD instruction of Figure 2 ("ADD 2 x,y"), which is sent, is modified to contain a pointer to a location within VERSION 1. This modified instruction is labeled INSTRUCTION SENT in Figure 1J. This pointer is "2", which indicates the "c".

The information sent (ie, the bytes representing "x" and "y") is then EX-ORed with the data beginning with the "c". Figure 1J, left side, indicates the EX-OR operation. The instruction sent is "ADD 2 2" plus the result of the EX-OR operation. This EX-OR result contains no information of value to an eavesdropper, because the eavesdropper has no access to VERSION 1.

The data contained in the received instruction, indicated on the right side of Figure 1K, is EX-ORed with the same data in VERSION 1, namely, that which begins with the "c". This EX-ORing recovers the original data, namely "x,y".

This procedure exploits a property of the EX-OR operation, namely, that EX-ORing a first word with a second word produces a third word. ED-ORing the third word with the second recovers the first.

Thus, from the transmitted instruction, which is

ADD 2 2 [EX-OR result],

one obtains

ADD 2 x,y,

which is the intended instruction.

Important Considerations

1. The order in which the COPY and ADD instructions occur is, of course, important. If they are scrambled as to order, a different VERSION 2 will be obtained.

From another perspective, the sequence itself is a type of information. The sequence can be viewed as a permutation of the instructions. The permutation contains information, just as permutations of the English alphabet create words, which convey information.

Of course, the instructions could be transmitted in a scrambled order, if information is included which allows reconstruction of the sequence. For example, each instruction could be numbered.

Irrespective of how execution of the proper sequence of instructions is attained, the execution produces a replica of VERSION 2, by a concatenation process. That is, returning to the example of Figures 2B through 2E:

1. First "abcd" is written into the replica (Figure 2B). "abcd" is taken from VERSION 1.
2. Then "xy" is concatenated (Figure 2C). "xy" is obtained from VERSION 1.
3. Then "xyxyxy" is concatenated (Figure 2D). "xyxyxy" is obtained from VERSION 2. (Alternately, four pairs of "xy" could have been taken from VERSION 1 and concatenated in step 2. This would be less efficient, however.)
4. Then "bcdef" is concatenated (Figure 2E). "bcdef " is obtained from VERSION 1.

2. Further regarding Consideration Number 1, above, the Inventors point out that each COPY and ADD instruction contains the length of the passage to be COPIED or ADDED. Thus, for any given COPY or ADD instruction, the address within VERSION 2 where the COPYING or ADDING occurs can be readily computed, based on the total preceding lengths. (The code of Figure 3 illustrates this, because the pointer is computed based on these lengths.)

3. The invention can be used to update any type of data file. It is not limited to text files, or to files of other specific types. It can handle binary files generally.

Files contain characters. Each character is commonly represented by a byte of data. Since a byte contains eight bits, 2^8, or 256, possible characters can be represented by one byte.

In "text" files, all of these possible combinations are not used. Only those representing alphanumerics, and some punctuation, are used.

In "binary" files, all possible 256 combinations are used. The invention can handle binary files. In contrast, differencing programs of the prior art can handle only text files.

4. The invention is not limited to remote replication of VERSION 2 from a stored VERSION 1. In addition, a later version of a program can be stored at a single location, by using the ADD and COPY instructions, rather than storing the version in its entirety. This approach will save storage space. When a version is desired to be reconstructed, the program of Figure 3 is executed.

5. It is not necessary that the version of a file which is reconstructed be the actual, chronologically later version of an earlier file. For example, VERSION 1 can be reconstructed from VERSION 2.

Technical Discussion

A more technical discussion of the code given in Figures 3 and 5 will now be given.

Overview

A data file can be considered as a sequence of bytes. On virtually all current computers, a byte is the smallest natural unit that can be compressed efficiently in terms of storage, communication, and in-memory data manipulation. Though the term "data file" often refers to a file when stored on-disk, the present invention also allows such a sequence of bytes to be some segment of main memory.

Figure 1 shows an example of using the present invention to keep data in synchronization between two computers. First, on the source computer, the two versions of the data, VERSION1 and VERSION2, are compared to produce a transformation that takes VERSION1 to VERSION2. This transformation is transmitted via some communications channel to the target computer. Then, on the target computer, the transformation and the local copy of VERSION1 are used to reconstruct VERSION2.

Coding and Decoding Transformation Instructions

The transformation computed by the present invention consists of a sequence of two types of instructions, COPY and ADD. During the reconstruction of VERSION2, the COPY instruction defines the position and length of some existing segment of data to be copied while the ADD instruction defines a segment of data to be added. Figure 2 shows an example of two data files in which VERSION1 consists of the sequence of bytes "abc dab cda bcd efgh" (spaces are added here for readability). VERSION2 consists of the sequence "abc dxy xyx yxy bcdef." Thus, VERSION1 has length 16, and VERSION2 has length 17. The transform shown in Figure 2 shows the instructions needed to reconstruct VERSION2 from VERSION1. Note that we use the convention that positions in VERSION1 are coded starting from 0, while positions in VERSION2 are coded starting from the length of VERSION1. For example, the third instruction of the transform in Figure 2 is a COPY instruction that copies 6 bytes starting from position 4 of VERSION2, which is coded as 20.

Figure 3 shows the procedure of how the reconstruction of a version is carried out in general. Line 1 initializes the variable "n" to the length of VERSION1. Line 2 sets the current position "c" in VERSION2 to zero. Line 3 starts a loop that is terminated on line 6 after the end-of-file condition is detected in line 5. Line 4 calls the function readinst() to read an instruction. Line 7 reads the copying size or the data size using the function readsize(). Lines 8 and 9 check to see if the instruction is an ADD instruction and, if so, reads the data into VERSION2 starting at the current position c. Lines 10 to 15 process a COPY instruction by reading in the position code, then performing the copy operation either from VERSION1 or VERSION2, as appropriate. Line 16 increases the current copy position in VERSION2 by the length of the newly reconstructed data. The copy() function is a simple function to copy data from one area of disk memory (or main memory) to another. However, the readinst(), readsize(), readpos(), and readdata() functions must be defined, based on some concrete definitions of how the COPY and ADD instructions and their parameters are encoded.

Applying the procedure of Figure 3 to the example of Figure 1, we see that there are four steps in the decoding. The first step copies four bytes "abcd" from VERSION1, starting at position 0. THe second step adds 2 data bytes "xy". The third step copies from VERSION2 6 bytes starting at its position 4 (counting from 0 and coded as 20 by our convention). Note that at the start of this step, only the first 6 bytes "abcdxy" of VERSION2 have been reconstructed as only the two bytes "xy" are available to copy. However, since data is copied from left to right, whenever a byte is to be copied it will have been constructed. The fourth and final step copies the 5 bytes "bcdef " from position 9 of VERSION1.

We now give a description of the encoding of COPY and ADD instructions. This particular embodiment of these instructions are chosen since in the Inventors' experimentation they perform well for many different types of data. GIven the description, the readinst(), readsize(), and readpos() functions above can be straightforwardly implemented. Each instruction is coded starting with a control byte. The eight bits of a control byte are divided into two parts. The first four bits represent numbers from 0 to 15, each of which defines a type of instruction and a coding of some auxiliary information. Below is an enumeration of the first 10 values of the first four bits:

0: an ADD instruction
1,2,3: a COPY instruction with position in the QUICK cache;
4: a COPY instruction with position coded as SELF;
5: a COPY instruction with position coded as difference from HERE;
6,7,8,9: a COPY instruction with position coded from a RECENT cache.

The QUICK cache is an array of size 768 (3 x 256). Each index of this array contains the value p of the position of a recent COPY instruction such that p modulo 768 is the array index. This cache is updated after each COPY instruction is output (during coding) or processed (during decoding). A COPY instruction of type 1, 2, or 3 will be immediately followed by a byte whose value is from 0 to 255 that must be added to 0, 256, or 512, respectively, to compute the array index where the actual position is stored.

A COPY instruction of type 4 has the copying position coded as a sequence of bytes.

A COPY instruction of type 5 has the difference between the copying position and the current position coded as a sequence of bytes.

The RECENT cache is an array having 4 indices, and stores the most recent 4 copying positions. Whenever a COPY instruction is output (during coding) or processed (during decoding), its copying position replaces the oldest position in the cache. A COPY instruction of type 6 (respectively 7, 8, 9) corresponds to cache index 1 (respectively 2, 3, and 4). Its copying position is guaranteed to be larger than the position stored in the corresponding cache index and only the difference is coded.

For the ADD instruction and COPY instruction of type from 1 through 9, the second 4 bits of the control byte, if not zero, code the size of the data involved. If these bits are 0, the respective size is coded as a subsequent sequence of bytes.

It is a result of the encoding method that an ADD instruction is never followed by another ADD instruction. Frequently, an ADD instruction has data size less than or equal to 4, and the following COPY instruction is also small. In such cases, it is advantageous to merge the two instructions into a single control byte. The values from 10 to 15 of the first 4 bits code such merged pairs of instructions. In such a case, the first 2 bits of the second 4 bits in the control byte code the size of the ADD instruction and the remaining 2 bits code the size of the COPY instruction. Below is an enumeration of the values from 10 to 15 of the first 4 bits:

10: a merged ADD/COPY instruction with copy position coded as SELF,

11: a merged ADD/COPY instruction with copy position coded as difference from HERE;

12,13,14,15: a merge ADD/COPY instruction with copy position coded from a RECENT cache.

Figure 4 shows an encoding of the four instructions of the transform shown in Figure 2. For each instruction, all eight bits of the control byte are shown. For example, the first four bits of the second control byte are zero, showing that the byte codes an ADD instruction. The second four bits of the same byte show that the size of the ADD instruction is 2. The two data bytes "xy" come after the control byte. All three COPY instructions are coded using the SELF type, meaning that their copying positions are coded in bytes (with shown values) following the control bytes. The size parameters of all the instructions are small enough so that they can all be encoded inside the control bytes. This small example shows that VERSION2 which is 17 bytes long can be coded in a transformation using only 9 bytes.

Fast Computation of Matched Segments

Figure 5 shows the method to partition VERSION2 into segments and encode them as COPY and ADD instructions. Note that, for this encoding method, matches of short lengths are not useful because they take up at least as much space to encode as the data that they match. So the matching method can be tuned to ignore short matches. The minimum match length that we use is 4 and represented by the variable MIN in the body of the encoding procedure.

Line 1 of Figure 5 initializes a search table T to empty. For efficiency, T is kept as a hash table with chaining for collisions. This data structure is standard, and is described in any data structure and algorithm textbook, such as pages 111 - 112 in "The Design and Analysis of Computer Algorithms" by A. Aho, J. Hopcroft, and J. Ullman, published by Addison-Wesley, 1974.

Table T contains certain selected positions in the two versions. These positions are inserted with the procedure insert() and used in the procedures search() and extend() for fast searching of longest matched data segments. Lines 2 and 3 call the procedure process() to select positions from VERSION1 and VERSION2 to insert into the table T. COPY and ADD instructions are also generated during the processing of VERSION2.

Lines 4 to 45 define the procedure process(). Lines 5 and 6 initialize the variables "n" and "m" to the length of VERSION1 and the version being processed. Line 7 initializes the curent position "c" of the version being processed to 0. Line 8 initializes the start of data for an ADD instruction to -1 (ie, none). Lines 9 and 10 initialize the position and length of the longest match for the data segment starting at position c Lines 11 to 42 define the main loop that process the given version. Lines 12 to 18 compute a longest, already-processed data segment that matches with the data segment starting at c. The procedure search() called on line 12 finds a match of length len+1. This procedure uses MIN bytes starting at position c+len-(MIN-1) as a key to search for a matching position in the table T. Then it matches backward the data in "seq" and the data in the appropriate version to see if the match covers everything from c to c+len+1.

If such a match is found, the procedure extend() called on line 16 extends the match as far forward as possible. Lines 14 and 15 correspond to the case when there is no match and we break out of the search loop. Otherwise, the loop is repeated to find a longer match. Lines 19 and 26 insert the current unmatched position c into the table T.

Note that the actual value to be inserted is c if seq is VERSION1; otherwise, it is c plus the length of VERSION1 (which is the convention for encoding positions in VERSION2 that we described earlier). The procedure insert(T,seq,p,origin) inserts the coded position p+origin into table T using as its key. MIN bytes starting from position p in version seq.

Lines 20 and 21 set the variable add to c, if it is not yet defined, to indicate that this is the beginning of a segment of unmatched data. Line 25 moves the current position forward by 1 for the next iteration of the processing loop. Lines 27 and 28 correspond to the event of finding a longest matched segment. Lines 28 and 29 call the procedure writeinst() to write out the COPY and ADD instructions according to the description given earlier.

The first two arguments to writeinst() define the data for an ADD instruction if add is zero or positive. The second two arguments define parameters for the COPY instruction. The working of this procedure is straightforward so we shall omit its description. Lines 30 to 36 insert into table T MIN-1 positions at the end of the matched data segment. Lines 37 and 38 increase c by the length of the match, and resets add to -1. Lines 40 and 41 terminate processing if there is not sufficient data to be processed. Lines 43 and 44 output any final unmatched data from VERSION2 as an ADD instruction.

Figure 6 shows a procedure to insert a position into the table T. Line 2 of Figure 6 indicates that the key is the MIN bytes of the sequence "seq" starting at the position p. Line 3 constructs the coded position. Line 4 inserts the pair (key, pos) into the table T.

Figure 7 shows a procedure to search for a match. Lines 3 and 4 construct the search key, which consists of MIN-1 bytes at the end of the current longest match length, plus a new unmatched byte. Lines 5 to 19 try to extend the match length if possible. This is done by looking at every element of the table T that matches the constructed search key and seeing if the part of the current match left to the key position matches with the corresponding part of the element being considered. This test is carried out on line 17. If this is true, then search() returns the starting position of the match on line 18. Line 20 returns "-1" to indicate that there is no match longer than "len".

After the call to search() is successful on line 13 of Figure 5, the match length is now known to be at least 1 more than the "len" value in Figure 5. Figure 8 shows the extend() procedure, which extends the match to the right as far as possible. Lines 2 to 10 set up the correct sequences to search for matches. lines 11 to 13 perform the extension. line 14 returns the entire matched length.

When applied to the example of Figure 2, the above method will compute exactly the set of instructions exhibited in the same figure. Figure 6 shows the positions in the example VERSION1 and VERSION2 that would be inserted into table T.

Security Enhancement

Given two versions, VERSION1 and VERSION2, and a computed transformation, only the ADD instructions in the transformation carry raw data from VERSION2. Such data can be utilized by a snooper to gain valuable information about VERSION2. In applications having high level of security requirements, to prevent leakage of information, the ADD instruction can be modified as follows. First, each ADD instruction is modified to also have a copying address which would be some position randomly selected from VERSION1 in such a way that the data segment starting at that position is at least as long as the length of the ADD data. If this is not possible, the ADD data can be broken into smaller units. Then, the raw data is xor-ed with data from such a selected segment of data before output to the transformation. For example, using the same ADD instruction in Figure 2, if the selected position in VERSION1 was 2, then this position would be output immediately after the control byte, and the two data bytes "xy" would be xor-ed with the two bytes "cd" before output.

On decoding, the same xor operation must be done to each data byte before outputting. This works because the mathematical property of xor guarantees that when a byte is xor-ed twice with another same byte, it retains its original value. However, we can now guarantee that a snooper will not be able to gain any information from the transformed data bytes unless he or she already has a copy of VERSION1, which was assumed to be secure.

Figure 7 shows the instruction of Figure 4 encoded using this more secure method. The ADD instruction now has a position 2. The data bytes "x" and "y" are shown being xor-ed with "c" and "d" respectively.

Tue Aug 30 09:21:37 1994

peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelta.h 1

APPENDIX
(24 pages)

```
#ifndef _VDELTA_H
#define _VDELTA_H 1

#ifndef __KPV__
#define __KPV__            1

#ifndef __STD_C
#ifdef __STDC__
#define   __STD_C          1
#else
#if __cplusplus
#define __STD_C            1
#else
#define __STD_C            0
#endif /*__cplusplus*/
#endif /*__STDC__*/
#endif /*__STD_C*/

#ifndef _BEGIN_EXTERNS_
#if __cplusplus
#define _BEGIN_EXTERNS_    extern "C" {
#define _END_EXTERNS_      }
#else
#define _BEGIN_EXTERNS_
#define _END_EXTERNS_
#endif
#endif /*_BEGIN_EXTERNS_*/

#ifndef _ARG_
#if __STD_C
#define _ARG_(x)  x
#else
#define _ARG_(x)  ()
#endif
#endif /*_ARG_*/

#ifndef Void_t
#if __STD_C
#define Void_t             void
#else
#define Void_t             char
#endif
#endif /*Void_t*/

#ifndef NIL
#define NIL(type) ((type)0)
#endif /*NIL*/

#endif /*__KPV__*/

/* user-supplied functions to do io */
typedef struct _vddisc_s  Vddisc_t;
typedef int(*     Vdio_f)_ARG_((int, Void_t*, int, long, Vddisc_t*));
struct _vddisc_s
{ Vdio_f  readf;           /* to read data         */
  Vdio_f  writef;          /* to write data        */
  long    window;          /* window size if any   */
};

```

Tue Aug 30 09:21:37 1994

peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelta.h   1

```
/* types that can be given to the IO functions */
#define VD_SOURCE 1       /* io on the source data        */
#define VD_TARGET 2       /* io on the target data        */
#define VD_DELTA  3       /* io on the delta data         */

/* magic header for delta output */
#define VD_MAGIC  "vd01"

_BEGIN_EXTERNS_
extern long       vddelta _ARG_((Void_t*, long, Void_t*, long, Vddisc_t*));
extern long       vdupdate _ARG_((Void_t*, long, Void_t*, long, Vddisc_t*));
_END_EXTERNS_

#endif /*_VDELTA_H*/
```

```
                    Wed Aug 10 21:24:44 1994
| peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelhdr.h    1

#ifndef _VDELHDR_H
#define _VDELHDR_H          1

#include  "vdelta.h"

#if __STD_C
#include  <stddef.h>
#else
#include  <sys/types.h>
#endif

#ifdef DEBUG
_BEGIN_EXTERNS_
extern int                  abort();
_END_EXTERNS_
#define ASSERT(p) ((p) ? 0 : abort())
#define DBTOTAL(t,v)        ((t) += (v))
#define DBMAX(m,v)          ((m) = (m) > (v) ? (m) : (v) )
#else
#define ASSERT(p)
#define DBTOTAL(t,v)
#define DBMAX(m,v)
#endif

/* short-hand notations */
#define reg                 register
#define uchar               unsigned char
#define uint                unsigned int
#define ulong               unsigned long

/* default window size - Chosen to suit malloc() even on 16-bit machines. */
#define MAXINT              ((int)(((uint)~0) >> 1))
#define MAXWINDOW ((int)(((uint)~0) >> 2))
#define DFLTWINDOW          (MAXWINDOW <= (1<<14) ? (1<<14) : (1<<16) )
#define HEADER(w) ((w)/4)

#define M_MIN               4        /* min number of bytes to match */

/* The hash function is s[0]*alpha^3 + s[1]*alpha^2 + s[2]*alpha + s[3] */
#define   ALPHA             33
#if 0
#define A1(x,t)             (ALPHA*(x))
#define A2(x,t)             (ALPHA*ALPHA*(x))
#define A3(x,t)             (ALPHA*ALPHA*ALPHA*(x))
#else     /* fast multiplication using shifts&adds */
#define A1(x,t)             ((t = (x)), (t + (t<<5)) )
#define A2(x,t)             ((t = (x)), (t + (t<<6) + (t<<10)) )
#define A3(x,t)             ((t = (x)), (t + (t<<5) + ((t+(t<<4))<<6) + ((t+(t<<4))<<11)
)
#endif
#define HINIT(h,s,t)        ((h = A3(s[0],t)), (h += A2(s[1],t)), (h += A1(s[2],t)+s[3])
)
#define HNEXT(h,s,t)        ((h -= A3(s[-1],t)), (h = A1(h,t) + s[3]) )

#define EQUAL(s,t)          ((s)[0] == (t)[0] && (s)[1] == (t)[1] && \
                     (s)[2] == (t)[2] && (s)[3] == (t)[3] )

/* Every instruction will start with a control byte.
** For portability, only 8 bits of the byte are used.
```

Wed Aug 10 21:24:44 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelhdr.h

```
** The bits are used as follows:
**          iiii ssss
** ssss: size of data involved.
** iiii: this defines 16 instruction types.
**          0: an ADD instruction.
**          1,2,3: COPY with K_QUICK addressing scheme.
**          4,5: COPY with K_SELF,K_HERE addressing schemes.
**          6,7,8,9: COPY with K_RECENT addressing scheme.
**                  For the above types, ssss if not zero codes the size.
**                  otherwise, the size is coded in subsequent bytes.
**          10,11: merged ADD/COPY with K_SELF,K_HERE addressing
**          12,13,14,15: merged ADD/COPY with K_RECENT addressing
**                  For merged ADD/COPY instructions, ssss is divided into "cc aa"
**                  where cc codes the size of COPY and aa codes the size of ADD
*/

#define VD_BITS             8       /* # bits usable in a byte              */

#define S_BITS              4       /* bits for the size field              */
#define I_BITS              4       /* bits for the instruction type        */

/* The below macros compute the coding for a COPY address.
** There are two caches, a "quick" cache of (K_QTYPE*256) addresses
** and a revolving cache of K_RTYPE "recent" addresses.
** First, we look in the quick cache to see if the address is there.
** If so, we use the cache index as the code.
** Otherwise, we compute from 0, the current location and
** the "recent" cache an address that is closest to the being coded address,
** then code the difference. The type is set accordingly.
**
** An invariance is 2*K_MERGE + K_QTYPE - 1 == 16
*/
#define K_RTYPE             4               /* # of K_RECENT types          */
#define K_QTYPE             3               /* # of K_QUICK types           */
#define K_MERGE             (K_RTYPE+2)     /* # of types allowing add-copy */
#define K_QSIZE             (K_QTYPE<<VD_BITS) /* size of K_QUICK cache     */

#define K_QUICK             1               /* start of K_QUICK types       */
#define K_SELF              (K_QUICK+K_QTYPE)
#define K_HERE              (K_SELF+1)
#define K_RECENT  (K_HERE+1)      /* start of K_RECENT types      */

#define K_DDECL(quick,recent,rhere)       /* cache decls in vdelta        */ \
  int quick[K_QSIZE]; int recent[K_RTYPE]; int rhere/*;*/
#define K_UDECL(quick,recent,rhere)       /* cache decls in vdupdate      */ \
  long quick[K_QSIZE]; long recent[K_RTYPE]; int rhere/*;*/
#define K_INIT(quick,recent,rhere) \
  { quick[rhere=0] = (1<<7); \
    while((rhere += 1) < K_QSIZE) quick[rhere] = rhere + (1<<7); \
    recent[rhere=0] = (1<<8); \
    while((rhere += 1) < K_RTYPE) recent[rhere] = (rhere+1)*(1<<8); \
  }
#define K_UPDATE(quick,recent,rhere,copy) \
  { quick[copy%K_QSIZE] = copy; \
    if((rhere += 1) >= K_RTYPE) rhere = 0; recent[rhere] = copy; \
  }

#define VD_ISCOPY(k)      ((k) > 0 && (k) < (K_RECENT+K_RTYPE) )
#define K_ISMERGE(k)      ((k) >= (K_RECENT+K_RTYPE))
```

Wed Aug 10 21:24:44 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelcdr.c 3

```
#define A_SIZE          ((1<<S_BITS)-1)         /* max local ADD size   */
#define A_ISLOCAL(s)    ((s) <= A_SIZE )        /* can be coded locally */
#define A_LPUT(s) (s)                   /* coded local value    */
#define A_PUT(s)  ((s) - (A_SIZE-1) )   /* coded normal value   */

#define A_ISHERE(i)     ((i) & A_SIZE)          /* locally coded size   */
#define A_LGET(i) ((i) - A_SIZE)
#define A_GET(s)  ((s) + (A_SIZE+1) )

#define C_SIZE          ((1<<S_BITS)+M_MIN-2)   /* max local COPY size  */
#define C_ISLOCAL(s)    ((s) <= C_SIZE )        /* can be coded locally */
#define C_LPUT(s) ((s) - (M_MIN-1) )    /* coded local value    */
#define C_PUT(s)  ((s) - (C_SIZE+1) )   /* coded normal value   */

#define C_ISHERE(i)     ((i) & ((1<<S_BITS)-1)) /* size was coded local */
#define C_LGET(i) (((i) & ((1<<S_BITS)-1)) + (M_MIN-1) )
#define C_GET(s)  ((s) + (C_SIZE+1) )

#define K_PUT(k)  ((k) << S_BITS)
#define K_GET(i)  ((i) >> S_BITS)

/* coding merged ADD/COPY instructions */
#define A_TINY          2               /* bits for tiny ADD            */
#define A_TINYSIZE      (1<<A_TINY)             /* max tiny ADD size    */
#define A_ISTINY(s)     ((s) <= A_TINYSIZE )
#define A_TPUT(s) ((s) - 1)
#define A_TGET(i) (((i) & (A_TINYSIZE-1)) + 1)

#define C_TINY          2               /* bits for tiny COPY           */
#define C_TINYSIZE      ((1<<C_TINY) + M_MIN-1) /* max tiny COPY size   */
#define C_ISTINY(s)     ((s) <= C_TINYSIZE)
#define C_TPUT(s) (((s) - M_MIN) << A_TINY)
#define C_TGET(i) ((((i) >> A_TINY) & ((1<<C_TINY)-1)) + M_MIN )

#define K_TPUT(k) (((k)+K_MERGE) << S_BITS)

#define MEMCPY(to,from,n) \
  switch(n) \
  { default: memcpy((Void_t*)to,(Void_t*)from,(size_t)n); \
             to += n; from += n; break; \
    case 7 : *to++ = *from++; \
    case 6 : *to++ = *from++; \
    case 5 : *to++ = *from++; \
    case 4 : *to++ = *from++; \
    case 3 : *to++ = *from++; \
    case 2 : *to++ = *from++; \
    case 1 : *to++ = *from++; \
    case 0 : break; \
  }

/* Below here is code for a buffered I/O subsystem to speed up I/O */
#define I_SHIFT         7
#define I_MORE          (1<<I_SHIFT)                    /* continuation bit     */
#define I_CODE(n) ((uchar)((n)&(I_MORE-1)) )    /* get lower bits       */

/* structure to do buffered IO */
typedef struct _vdio_s
{ uchar*          next;
  uchar*          endb;
  Vddisc_t*       disc;
```

Wed Aug 10 21:24:44 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelhdr.h 4

```
  long          here;
  uchar         buf[1024];
} Vdio_t;

#define READF(io) ((io)->disc->readf)
#define WRITEF(io)        ((io)->disc->writef)
#define BUF(io)           ((io)->buf)
#define BUFSIZE(io)       sizeof((io)->buf)
#define NEXT(io)  ((io)->next)
#define ENDB(io)  ((io)->endb)
#define DISC(io)  ((io)->disc)
#define HERE(io)  ((io)->here)
#define RINIT(io,disc)    ((io)->endb = ((io)->next = (io)->buf), \
                         (io)->here = 0, (io)->disc = (disc) )
#define WINIT(io,disc)    ((io)->endb = ((io)->next = (io)->buf) + BUFSIZE(io), \
                         (io)->here = 0, (io)->disc = (disc) )
#define REMAIN(io)        (ENDB(io) - NEXT(io))
#define VDPUTC(io,c)      ((REMAIN(io) > 0 || (*_Vdflsbuf)(io) > 0) ? \
                          (int)(*(io)->next++ = (c)) : -1 )
#define VDGETC(io)        ((REMAIN(io) > 0 || (*_Vdfilbuf)(io) > 0) ? \
                          (int)(*(io)->next++) : -1 )

typedef struct _vdbufio_s
{ int(*   vdfilbuf)_ARG_((Vdio_t*));
  int(*   vdflsbuf)_ARG_((Vdio_t*));
  ulong(* vdgetu)_ARG_((Vdio_t*, ulong));
  int(*   vdputu)_ARG_((Vdio_t*, ulong));
  int(*   vdread)_ARG_((Vdio_t*, uchar*, int));
  int(*   vdwrite)_ARG_((Vdio_t*, uchar*  int));
} Vdbufio_t;
#define _Vdfilbuf _Vdbufio.vdfilbuf
#define _Vdflsbuf _Vdbufio.vdflsbuf
#define _Vdgetu           _Vdbufio.vdgetu
#define _Vdputu           _Vdbufio.vdputu
#define _Vdread           _Vdbufio.vdread
#define _Vdwrite  _Vdbufio.vdwrite

_BEGIN_EXTERNS_
extern Vdbufio_t  _Vdbufio;
extern Void_t*            memcpy _ARG_((Void_t*, const Void_t*, size_t));
extern Void_t*            malloc _ARG_((size_t));
extern void               free _ARG_((Void_t*));
_END_EXTERNS_

#endif /*_VDELHDR_H*/
```

Wed Sep 28 08:36:11 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdelta.c

```
#include  "vdelhdr.h"

/*        Compute a transformation that takes source data to target data
**
**        Written by (Kiem-)Phong Vo, kpv@research.att.com, 5/20/94
*/

#ifdef DEBUG
long      S_copy, S_add;  /* amount of input covered by COPY and ADD       */
long      N_copy, N_add;  /* # of COPY and ADD instructions                */
long      M_copy, M_add;  /* max size of a COPY or ADD instruction         */
long      N_merge;        /* # of merged instructions                      */
#endif

#define MERGABLE(a,c,k) ((a) > 0 && A_ISTINY(a) && \
                         (c) > 0 && C_ISTINY(c) && \
                         (k) >= K_SELF )

typedef struct _match_s   Match_t;
typedef struct _table_s   Table_t;
struct _match_s
{ Match_t*        next;           /* linked list ptr      */
};
struct _table_s
{ Vdio_t          io;             /* io structure         */
  uchar*          src;            /* source string        */
  int             n_src;
  uchar*          tar;            /* target string        */
  int             n_tar;
  K_DDECL(quick,recent,rhere);    /* address caches       */
  Match_t*        base;           /* base of elements     */
  int             size;           /* size of hash table   */
  Match_t**       table;          /* hash table           */
};

/* encode and output delta instructions */
#if __STD_C
static vdputinst(Table_t* tab, uchar* begs, uchar* here, Match_t* match, int n_copy
#else
static vdputinst(tab, begs, here, match, n_copy)
Table_t*  tab;
uchar*              begs;   /* ADD data if any      */
uchar*              here;   /* current location     */
Match_t*  match;  /* best match if any    */
int                 n_copy; /* length of match      */
#endif
{
   reg int n_add, i_add, i_copy, k_type;
   reg int n, c_addr, copy, best, d;

   n_add = begs ? here-begs : 0;            /* add size             */
   c_addr = (here-tab->tar)+tab->n_src;     /* current address      */
   k_type = 0;

   if(match)       /* process the COPY instruction */
   {       /**/DBTOTAL(N_copy,1); DBTOTAL(S_copy,n_copy); DBMAX(M_copy,n_copy);

           best = copy = match - tab->base;
           k_type = K_SELF;
```

```
                if((d = c_addr - copy) < best)
                {       best = d;
                        k_type = K_HERE;
                }
                for(n = 0; n < K_RTYPE; ++n)
                {       if((d = copy - tab->recent[n]) < 0    d >= best)
                                continue;
                        best = d;
                        k_type = K_RECENT+n;
                }
                if(best >= I_MORE && tab->quick[n = copy%K_QSIZE] == copy)
                {       for(d = K_QTYPE-1; d > 0; --d)
                                if(n >= (d<<VD_BITS) )
                                        break;
                        best = n - (d<<VD_BITS); /**/ASSERT(best < (1<<VD_BITS));
                        k_type = K_QUICK+d;
                }

                /**/ASSERT(best >= 0);
                /**/ASSERT((k_type+K_MERGE) < (1<<I_BITS) );

                /* update address caches */
                K_UPDATE(tab->quick,tab->recent,tab->rhere,copy);

                /* see if mergable to last ADD instruction */
                if(MERGABLE(n_add,n_copy k_type) )
                {       /**/DBTOTAL(N_merge,1);
                        i_add = K_TPUT(k_type) A_TPUT(n_add),C_TPUT(n_copy);
                }
                else
                {       i_copy = K_PUT(k_type);
                        if(C_ISLOCAL(n_copy) )
                                i_copy |= C_LPUT(n_copy);
                }
        }

        if(n_add > 0)
        {       /**/DBTOTAL(N_add,1); DBTOTAL(S_add,n_add); DBMAX(M_add,n_add);

                if(!MERGABLE(n_add,n_copy,k_type) )
                        i_add = A_ISLOCAL(n_add) ? A_LPUT(n_add) : 0;

                if(VDPUTC((Vdio_t*)tab,i_add) < 0 )
                        return -1;
                if(!A_ISLOCAL(n_add) &&
                   (*_Vdputu)((Vdio_t*)tab, (ulong)A_PUT(n_add)) < 0 )
                        return -1;
                if((*_Vdwrite)((Vdio_t*)tab, begs, n_add) < 0 )
                        return -1;
        }

        if(n_copy > 0)
        {       if(!MERGABLE(n_add,n_copy,k_type) && VDPUTC((Vdio_t*)tab,i_copy) < 0
)
                        return -1;

                if(!C_ISLOCAL(n_copy) &&
                   (*_Vdputu)((Vdio_t*)tab, (ulong)C_PUT(n_copy)) < 0 )
                        return -1;

                if(k_type >= K_QUICK && k_type < (K_QUICK+K_QTYPE) )
                {       if(VDPUTC((Vdio_t*)tab,(uchar)best) < 0 )
                                return -1;
                }
                else
                {       if((*_Vdputu)((Vdio_t*)tab, (ulong)best) < 0 )
                                return -1;
                }
        }
        else
        {       if((*_Vdflsbuf)((Vdio_t*)tab) < 0)
                        return -1;
        }

        return 0;
}


/* Fold a string */
#if __STD_C
static vdfold(Table_t* tab, int output)
#else
static vdfold(tab, output)
Table_t*  tab;
int             output;
#endif
{
        reg ulong       key, n;
        reg uchar       *s, *sm, *ends, *ss, *heade;
        reg Match_t     *m, *list, *curm, *bestm;
        reg uchar       *add, *endfold;
        reg int         head, len, n_src = tab->n_src;
        reg int         size = tab->size;
        reg uchar       *src = tab->src, *tar = tab->tar;
        reg Match_t     *base = tab->base, **table = tab->table;

        if(!output)
        {       if(tab->n_src < M_MIN)
                        return 0;
                endfold = (s = src) + tab->n_src;
                curm = base;
        }
        else
        {       endfold = (s = tar) + tab->n_tar;
                curm = base+n_src;
                if(tab->n_tar < M_MIN)
                        return vdputinst(tab,s,endfold,NIL(Match_t*),0);
        }

        add = NIL(uchar*);
        bestm = NIL(Match_t*);
        len = M_MIN-1;
        HINIT(key,s,n);
        for(;;)
        {       for(;;) /* search for the longest match */
                {       if(!(m = table[key&size]) )
                                goto endsearch;
                        list = m = m->next;     /* head of list */

                        if(bestm) /* skip over past elements */
```

Wed Sep 28 08:36:11 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vddelta.c 4

```
			{	for(;;)
				{	if(m >= bestm-len)
						break;
					if((m = m->next) == list)
						goto endsearch;
				}
			}

			head = len - (M_MIN-1); /* header before the match */
			heade = s+head;
			for(;;)
			{	if((n = m-base) < n_src)
				{	if(n < head)
						goto next;
					sm = src - n;
				}
				else
				{	if((n -= n_src) < head)
						goto next;
					sm = tar - n;
				}

				/* make sure that the M_MIN bytes match */
				if(!EQUAL(heade,sm))
					goto next;

				/* make sure this is a real match */
				for(sm -= head, ss = s; ss < heade; )
					if(*sm++ != *ss++)
						goto next;
				ss += M_MIN;
				sm += M_MIN;
				ends = endfold;
				if((m-base) < n_src && (n = (src+n_src)-sm) < (ends-s
))
					ends = s+n;
				for(; ss < ends; ++ss, ++sm)
					if(*sm != *ss)
						goto extend;
				goto extend;

			next:	if((m = m->next) == list )
					goto endsearch;
			}

		extend:	bestm = m-head;
			n = len;
			len = ss-s;
			if(ss >= endfold)	/* already match everything */
				goto endsearch;

			/* check for a longer match */
			ss -= M_MIN-1;
			if(len == n+1)
				HNEXT(key,ss,n);
			else	HINIT(key,ss,n);
		}

	endsearch:
		if(bestm)
		{	if(output && vdputinst(tab,add,s,bestm,len) < 0)
				return -1;

			/* add a sufficient number of suffices */
			ends = (s += len);
			ss = ends - (M_MIN-1);
			if(!output)
				curm = base + (ss-src);
			else	curm = base + n_src + (ss-tar);

			len = M_MIN-1;
			add = NIL(uchar*);
			bestm = NIL(Match_t*);
		}
		else
		{	if(!add)
				add = s;
			ss = s;
			ends = (s += 1);	/* add one prefix */
		}

		if(ends > (endfold - (M_MIN-1)) )
			ends = endfold - (M_MIN-1);

		if(ss < ends) for(;;)	/* add prefices/suffices */
		{	n = key&size;
			if(!(m = table[n]) )
				curm->next = curm;
			else
			{	curm->next = m->next;
				m->next = curm;
			}
			table[n] = curm++;

			if((ss += 1) >= ends)
				break;
			HNEXT(key,ss,n);
		}

		if(s > endfold-M_MIN)	/* too short to match */
		{	if(!add)
				add = s;
			break;
		}

		HNEXT(key,s,n);
	}

	if(output)	/* flush output */
		return vdputinst(tab,add,endfold,NIL(Match_t*),0);
	return 0;
}


#if __STD_C
long vddelta(Void_t* src, long n_src, Void_t* tar, long n_tar, Vddisc_t* disc)
#else
long vddelta(src, n_src, tar, n_tar, disc)
Void_t*		src;	/* source string if not NULL	*/
long		n_src;	/* length of source data	*/
```

Wed Sep 28 08:36:11 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vddelta.c

```
Void_t*          tar;    /* target string if not NULL     */
long             n_tar;  /* length of target data         */
Vddisc_t* disc;  /* IO discipline                 */
#endif
{
	reg int         size, k, n, window;
	reg long        p;
	Table_t         tab;

	if(!disc || !disc->writef ||
	   (!disc->readf && ((n_tar > 0 && !tar) || (n_src > 0 && !src) ) ) )
		return -1;

	if(n_tar < 0)
		return -1;
	if(n_src < 0)
		n_src = 0;

	tab.n_src = tab.n_tar = tab.size = 0;
	tab.tar = tab.src = NIL(uchar*);
	tab.base = NIL(Match_t*);
	tab.table = NIL(Match_t**);
	WINIT(&tab.io,disc);

	if(disc->window <= 0)
		window = DFLTWINDOW;
	else if(disc->window > MAXWINDOW)
		window = MAXWINDOW;
	else    window = (int)disc->window;
	if((long)window > n_tar)
		window = (int)n_tar;
	if(n_src > 0 && (long)window > n_src)
		window = (int)n_src;

	/* try to allocate working space */
	while(window > 0)
	{       /* space for the target string */
		size = (n_tar == 0 || tar) ? 0 : window;
		if((long)size > n_tar)
			size = (int)n_tar;
		if(size > 0 && !(tab.tar = (uchar*)malloc(size*sizeof(uchar))) )
			goto reduce_window;

		/* space for sliding header or source string */
		if(n_src <= 0)  /* compression only */
			size = tar ? 0 : HEADER(window);
		else            /* differencing */
		{       size = src ? 0 : window;
			if((long)size > n_src)
				size = (int)n_src;
		}
		if(size > 0 && !(tab.src = (uchar*)malloc(size*sizeof(uchar))) )
			goto reduce_window;

		/* space for the hash table elements */
		size = window < n_tar ? window : (int)n_tar;
		if(n_src <= 0)
			size += window < n_tar ? HEADER(window) : 0;
		else    size += window < n_src ? window : (int)n_src;
		if(!(tab.base = (Match_t*)malloc(size*sizeof(Match_t))) )
```

```
                        goto reduce_window;

                /* space for the hash table */
                n = size/2;
                do (size = n); while((n &= n-1) != 0);
                if(size < 64)
                        size = 64;
                while(!(tab.table = (Match_t**)malloc(size*sizeof(Match_t*))) )
                        if((size >>= 1) <= 0)
                                goto reduce_window;

                /* if get here, successful */
                tab.size = size-1;
                break;

        reduce_window:
                if(tab.tar)
                {       free((Void_t*)tab.tar);
                        tab.tar = NIL(uchar*);
                }
                if(tab.src)
                {       free((Void_t*)tab.src);
                        tab.src = NIL(uchar*);
                }
                if(tab.base)
                {       free((Void_t*)tab.base);
                        tab.base = NIL(Match_t*);
                }
                if((window >>= 1) <= 0)
                        return -1;
        }

        /* amount processed */
        n = 0;

        /* output magic bytes and sizes */
        for(k = 0; VD_MAGIC[k]; k++)
                ;
        if((*_Vdwrite)(&tab.io,(uchar*)VD_MAGIC,k) != k ||
           (*_Vdputu)(&tab.io,(ulong)n_tar) <= 0 ||
           (*_Vdputu)(&tab.io,(ulong)n_src) <= 0 ||
           (*_Vdputu)(&tab.io,(ulong)window) <= 0 )
                goto done;

        /* do one window at a time */
        while(n < n_tar)
        {       /* prepare the source string */
                if(n_src <= 0)  /* data compression */
                {       if(n <= 0)
                                tab.n_src = 0;
                        else
                        {       size = HEADER(window);
                                if(tar)
                                        tab.src = tab.tar + tab.n_tar - size;
                                else    memcpy((Void_t*)tab.src,
                                            (Void_t*)(tab.tar + tab.n_tar - size),
                                            size );
                                tab.n_src = size;
                        }
                }
```

Wed Sep 28 08:36:11 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vddelta.c   8

```
		else	/* data differencing */
		{	if(n < n_src)
			{	if(n+window > n_src)
					p = n_src-window;
				else	p = n;
				if(src)
					tab.src = (uchar*)src + p;
				else
				{	size = (*disc->readf)(VD_SOURCE, tab.src,
								window, p, disc);
					if(size != window)
						goto done;
				}
			} /* else use last window */

			tab.n_src = window;
		}

		/* prepare the target string */
		size = (n_tar-n) < window ? (int)(n_tar-n) : window;
		tab.n_tar = size;
		if(tar)
			tab.tar = (uchar*)tar + n;
		else
		{	size = (*disc->readf)(VD_TARGET, tab.tar, size, (long)n, disc
);
			if((long)size != tab.n_tar)
				goto done;
		}

		/* reinitialize table before processing */
		for(k = tab.size; k >= 0; --k)
			tab.table[k] = NIL(Match_t*);
		K_INIT(tab.quick,tab.recent,tab.rhere);

		if(tab.n_src > 0 && vdfold(&tab,0) < 0)
			goto done;
		if(vdfold(&tab,1) < 0)
			goto done;

		n += size;
	}

done:
	if(!tar && tab.tar)
		free((Void_t*)tab.tar);
	if(tab.src && ((n_src <= 0 && !tar) || (n_src > 0 && !src) ) )
		free((Void_t*)tab.src);
	if(tab.base)
		free((Void_t*)tab.base);
	if(tab.table)
		free((Void_t*)tab.table);

	return n;
}
```

```
                    Sun Aug 14 11:55:45 1994

      peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c      1

#include  "vdelhdr.h"


/*        Apply the transformation source->target to reconstruct target
**        This code is designed to work even if the local machine has
**        word size smaller than that of the machine where the delta
**        was computed. A requirement is that "long" on the local
**        machine must be large enough to hold source and target sizes
**        It is also assumed that if an array is given, the size of
**        that array in bytes must be storable in an "int". This is
**        used in various cast from "long" to "int".
**
**        Written by (Kiem-)Phong Vo, kpv@research.att.com, 5/20/94
*/
typedef struct _table_s
{ Vdio_t          io;               /* io structure                 */
  uchar*          src;              /* source string                */
  long            n_src;
  uchar*          tar;              /* target string                */
  long            n_tar;
  long            s_org;            /* start of window in source    */
  long            t_org;            /* start of window in target    */
  uchar           data[128];        /* buffer for data transferring */
  char            s_alloc;          /* 1 if source was allocated    */
  char            t_alloc;          /* 1 if target was allocated    */
  char            compress;         /* 1 if compressing only        */
  K_UDECL(quick,recent,rhere);      /* address caches               */
} Table_t;

#if __STD_C
static vdunfold(Table_t* tab)
#else
static vdunfold(tab)
Table_t*  tab;
#endif
{
  reg long        size, copy;
  reg int         inst, k_type, n, r;
  reg uchar       *tar, *src, *to, *fr;
  reg long        t, c_addr, n_tar, n_src;
  reg Vdio_f      readf, writef;
  reg Vddisc_t*   disc;

  n_tar = tab->n_tar;
  tar = tab->tar;
  n_src = tab->n_src;
  src = tab->src;

  disc = tab->io.disc;
  readf = disc->readf;
  writef = disc->writef;

  for(t = 0, c_addr = n_src; t < n_tar; )
  {       if((inst = VDGETC((Vdio_t*)tab)) < 0)
                  return -1;
          k_type = K_GET(inst);

          if(!VD_ISCOPY(k_type))
          {       if(K_ISMERGE(k_type))   /* merge/add instruction        */
```

```
                        Sun Aug 14 11:55:45 1994

 peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c    2

                        size = A_TGET(inst);
                else if(A_ISHERE(inst)) /* locally coded ADD size       */
                        size = A_LGET(inst);
                else                    /* non-local ADD size           */
                {       if((size = VDGETC((Vdio_t*)tab)) < 0)
                                return -1;
                        if(size >= I_MORE &&
                            (size = (long)(*_Vdgetu)((Vdio_t*)tab,size)) < 0)
                                return -1;
                        size = A_GET(size);
                }
                if((t+size) > n_tar)    /* out of sync  */
                        return -1;
                c_addr += size;

                /* copy data from the delta stream to target */
                for(;;)
                {       if(!tar)
                        {       if((long)(n = sizeof(tab->data)) > size)
                                        n = (int)size;
                                if((*_Vdread)((Vdio_t*)tab,tab->data,n) != n
)
                                        return -1;
                                r = (*writef)(VD_TARGET,
                                        (Void_t*)tab->data, n,
                                        tab->t_org+t, disc);
                                if(r != n)
                                        return -1;
                        }
                        else
                        {       n = (int)size;
                                if((*_Vdread)((Vdio_t*)tab,tar+t,n) != n)
                                        return -1;
                        }
                        t += n;
                        if((size -= n) <= 0)
                                break;
                }

                if(K_ISMERGE(k_type))
                {       size = C_TGET(inst);
                        k_type -= K_MERGE;
                        goto do_copy;
                }
        }
        else
        {       if(C_ISHERE(inst))      /* locally coded COPY size */
                        size = C_LGET(inst);
                else
                {       if((size = VDGETC((Vdio_t*)tab)) < 0)
                                return -1;
                        if(size >= I_MORE &&
                            (size = (long)(*_Vdgetu)((Vdio_t*)tab,size)) < 0)
                                return -1;
                        size = C_GET(size);
                }
        do_copy:
                if((t+size) > n_tar)    /* out of sync */
                        return -1;

```

Sun Aug 14 11:55:45 1994

peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c

```
if((copy = VDGETC((Vdio_t*)tab)) < 0)
        return -1;
if(k_type >= K_QUICK && k_type < (K_QUICK+K_QTYPE) )
        copy = tab->quick[copy + ((k_type-K_QUICK)<<VD_BITS)]
)
else
{       if(copy >= I_MORE &&
           (copy = (long)(*_Vdgetu)((Vdio_t*)tab,copy)) < 0)
                return -1;
        if(k_type >= K_RECENT && k_type < (K_RECENT+K_RTYPE)
)
                copy += tab->recent[k_type - K_RECENT];
        else if(k_type == K_HERE)
                copy = c_addr - copy;
        /* else k_type == K_SELF */
}
K_UPDATE(tab->quick,tab->recent,tab->rhere,copy);
c_addr -= size;

if(copy < n_src)        /* copy from source data */
{       if((copy+size) > n_src) /* out of sync */
                return -1;
        if(src)
        {       n = (int)size;
                fr = src+copy;
                if(tar)
                {       to = tar+t;
                        MEMCPY(to,fr,n);
                }
                else
                {       r = (*writef)(VD_TARGET, (Void_t*)fr,
n,
                                tab->t_org+t, disc);
                        if(r != n)
                                return -1;
                }
                t += n;
        }
        else
        {       if(tab->compress)
                {       copy += tab->t_org - tab->n_src;
                        inst = VD_TARGET;
                }
                else
                {       copy += tab->s_org;
                        inst = VD_SOURCE;
                }
                for(;;)
                {       if(tar)
                        {       n = (int)size;
                                r = (*readf)(inst,
                                        (Void_t*)(tar+t), n,
                                        copy, disc);
                        }
                        else
                        {       n = sizeof(tab->data);
                                if((long)n > size)
                                        n = (int)size;
                                r = (*readf)(inst,
                                        (Void_t*)tab->data, n,
```

Sun Aug 14 11.55:45 1994

peregrine.zoo att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c   4

```
                                                                copy, disc );
                                                        if(r != n)
                                                                return -1;
                                                        r = (*writef)(VD_TARGET,
                                                                (Void_t*)tab->data, n
,
                                                                tab->t_org-t, disc);
                                                }
                                                if(r != n)
                                                        return -1;
                                                t += n;
                                                if((size -= n) <= 0)
                                                        break;
                                                copy += n;
                                        }
                                }
                        }
                        else    /* copy from target data */
                        {       copy -= n_src;
                                if(copy >= t || (copy+size) > n_tar) /* out-of-sync *
/
                                        return -1;
                                for(;;) /* allow for copying overlapped data */
                                {       reg long        s, a;
                                        if((s = t-copy) > size)
                                                s = size;
                                        if(tar)
                                        {       to = tar+t; fr = tar+copy; n = (int)s
;
                                                MEMCPY(to,fr,n);
                                                t += n;
                                                goto next;
                                        }

                                        /* hard read/write */
                                        a = copy;
                                        for(;;)
                                        {       if((long)(n = sizeof(tab->data)) > s)
                                                        n = (int)s;
                                                r = ( *readf)(VD_TARGET,
                                                        (Void_t*)tab->data, n,
                                                        a + tab->t_org, disc );
                                                if(r != n)
                                                        return -1;
                                                r = (*writef)(VD_TARGET,
                                                        (Void_t*)tab->data, n,
                                                        t + tab->t_org, disc );
                                                if(r != n)
                                                        return -1;
                                                t += n;
                                                if((s -= n) <= 0)
                                                        break;
                                                a += n;
                                        }

                                next:   if((size -= s) == 0)
                                                break;
                                }
                        }
                }
```

Sun Aug 14 11:55:45 1994

peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c

```
    }

    return 0;
}

#if __STD_C
long vdupdate(Void_t* src, long n_src, Void_t* tar, long n_tar, Vddisc_t* disc)
#else
long vdupdate(src, n_src, tar, n_tar, disc)
Void_t*         src;    /* source string if any */
long            n_src;  /* length of src        */
Void_t*         tar;    /* target space if any  */
long            n_tar;  /* size of tar          */
Vddisc_t* disc;
#endif
{
    Table_t tab;
    uchar   *data, magic[8];
    int     n, r;
    long    t, p, window;
    Vdio_f  readf, writef;

    if(!disc || !disc->readf || (!tar && !disc->writef) )
            return -1;
    readf = disc->readf;
    writef = disc->writef;

    /* initialize I/O buffer */
    RINIT(&tab.io,disc);

    /* check magic header */
    data = (uchar*)(VD_MAGIC);
    for(n = 0; data[n]; ++n)
            ;
    if((*_Vdread)(&tab.io,magic,n) != n)
            return -1;
    for(n -= 1; n >= 0; --n)
            if(data[n] != magic[n])
                    return -1;

    /* get true target size */
    if((t = (long)(*_Vdgetu)(&tab.io,0)) < 0 || (tar && n_tar != t) )
            return -1;
    n_tar = t;

    /* get true source size */
    if((t = (long)(*_Vdgetu)(&tab.io,0)) < 0 || (src && n_src != t) )
            return -1;
    n_src = t;

    /* get window size */
    if((window = (long)(*_Vdgetu)(&tab.io,0)) < 0)
            return -1;

    tab.compress = n_src == 0 ? 1 : 0;

    /* if we have space, it'll be faster to unfold */
    tab.tar = tab.src = NIL(uchar*);
    tab.t_alloc = tab.s_alloc = 0;

```

Sun Aug 14 11:53:45 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c

```
	if(n_tar > 0 && !tar && window < (long)MAXINT)
		n = (int)window;
	else	n = 0;
	if(n > 0 && (tab.tar = (uchar*)malloc(n*sizeof(uchar))) )
		tab.t_alloc = 1;

	if(n_src > 0 && !src && window < (long)MAXINT)
		n = (int)window;
	else if(n_src == 0 && window < n_tar && !tar && HEADER(window) < (long)MAXINT
)
		n = (int)HEADER(window);
	else	n = 0;
	if(n > 0 && (tab.src = (uchar*)malloc(n*sizeof(uchar))) )
		tab.s_alloc = 1;

	for(t = 0; t < n_tar; )
	{	tab.t_org = t;	/* current location in target stream */

		if(n_src == 0)	/* data compression */
		{	tab.s_org = 0;

			if(t == 0)
				tab.n_src = 0;
			else
			{	tab.n_src = HEADER(window);
				p = t - tab.n_src;
				if(tar)
					tab.src = (uchar*)tar + p;
				else if(tab.src)
				{	n = (int)tab.n_src;
					if(tab.tar)
					{	data = tab.tar + tab.n_tar - n;
						memcpy((Void_t*)tab.src,(Void_t*)data
,n);

					}
					else
					{	r = (*readf)(VD_TARGET,tab.src,n,p,di
sc);

						if(r != n)
							goto done;
					}
				}
			}
		}
		else	/* data differencing */
		{	tab.n_src = window;
			if(t < n_src)
			{	if((t+window) > n_src)
					p = n_src-window;
				else	p = t;

				tab.s_org = p;

				if(src)
					tab.src = (uchar*)src + p;
				else if(tab.src)
				{	n = (int)tab.n_src;
					r = (*readf)(VD_SOURCE,tab.src,n,p,disc);
					if(r != n)
						goto done;
```

Sun Aug 14 11:55:45 1994

peregrine.zoo.att.com!/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdupdate.c

```
                                }
                        }
                }

                if(tar)
                        tab.tar = (uchar*)tar+t;
                tab.n_tar = window < (n_tar-t) ? window : (n_tar-t);

                K_INIT(tab.quick,tab.recent,tab.there);
                if(vdunfold(&tab) < 0)
                        goto done;
                if(!tar && tab.tar)
                {       p = (*writef)(VD_TARGET,(Void_t*)tab.tar,(int)tab.n_tar,t,dis
c);
                        if(p != tab.n_tar)
                                goto done;
                }

                t += tab.n_tar;
        }

done:
        if(tab.t_alloc)
                free((Void_t*)tab.tar);
        if(tab.s_alloc)
                free((Void_t*)tab.src);

        return t;
}
```

Sun Aug 14 11:54:34 1994

peregrine.zoo.att.com:/n/gryphon/g7/kpv/Software/src/lib/vdelta/vdio.c

```
#include  "vdelhdr.h"


#if __STD_C
static _vdfilbuf(reg Vdio_t* io)
#else
static _vdfilbuf(io)
reg Vdio_t*       io;
#endif
{ reg int n;

  if((n = (*READF(io))(VD_DELTA,BUF(io),BUFSIZE(io),HERE(io),DISC(io))) > 0)
  {       ENDB(io) = (NEXT(io) = BUF(io)) + n;
          HERE(io) += n;
  }
  return n;
}

#if __STD_C
static _vdflsbuf(reg Vdio_t* io)
#else
static _vdflsbuf(io)
reg Vdio_t*       io;
#endif
{ reg int n;

  if((n = NEXT(io) - BUF(io)) > 0 &&
     (*WRITEF(io))(VD_DELTA,BUF(io),n,HERE(io),DISC(io)) != n)
          return -1;

  HERE(io) += n;
  NEXT(io) = BUF(io);
  return BUFSIZE(io);
}

#if __STD_C
static ulong _vdgetu(reg Vdio_t* io, reg ulong v)
#else
static ulong _vdgetu(io,v)
reg Vdio_t*       io;
reg ulong v;
#endif
{ reg int         c;

  for(v &= I_MORE-1;;)
  {       if((c = VDGETC(io)) < 0)
                  return (ulong)(-1L);
          if(!(c&I_MORE) )
                  return ((v<<I_SHIFT) | c);
          v = (v<<I_SHIFT) | (c & (I_MORE-1));
  }
}

#if __STD_C
static _vdputu(reg Vdio_t* io, ulong v)
#else
static _vdputu(io, v)
reg Vdio_t*       io;
reg ulong v;
```

Sun Aug 14 11:54:34 1994

peregrine.zoo.att.com./n/gryphon/g7/kpv/Software/src/lib/vdelta/vdio.c   2

```
#endif
{
  reg uchar       *s, *next;
  reg int         len;
  uchar           c[sizeof(ulong)+1];

  s = next = &c[sizeof(c)-1];
  *s = I_CODE(v);
  while((v >>= I_SHIFT) )
          *--s = I_CODE(v)|I_MORE;
  len = (next-s) + 1;

  if(REMAIN(io) < len && _vdflsbuf(io) <= 0)
          return -1;

  next = io->next;
  switch(len)
  {
  default: memcpy((Void_t*)next,(Void_t*)s,len); next += len; break;
  case 3: *next++ = *s++;
  case 2: *next++ = *s++;
  case 1: *next++ = *s;
  }
  io->next = next;

  return len;
}

#if __STD_C
static _vdread(Vdio_t* io, reg uchar* s, reg int n)
#else
static _vdread(io, s, n)
Vdio_t*          io;
reg uchar*       s;
reg int          n;
#endif
{
  reg uchar*      next;
  reg int         r, m;

  for(m = n; m > 0; )
  {       if((r = REMAIN(io)) <= 0 && (r = _vdfilbuf(io)) <= 0)
                  break;
          if(r > m)
                  r = m;

          next = io->next;
          MEMCPY(s,next,r);
          io->next = next;

          m -= r;
  }
  return n-m;
}

#if __STD_C
static _vdwrite(Vdio_t* io, reg uchar* s, reg int n)
#else
static _vdwrite(io, s, n)
Vdio_t*          io;
```

```
                              Sun Aug 14 11:54:34 1994
           peregrine.zoo.att.com!/n/gryphon/g7/kpv/software/src/lib/vdelta/vdio.c   3

reg uchar*        s;
reg int           n;
#endif
{
  reg uchar*      next;
  reg int         w, m;

  for(m = n; m > 0; )
  {       if((w = REMAIN(io)) <= 0 && (w = _vdflsbuf(io)) <= 0)
                  break;
          if(w > m)
                  w = m;

          next = io->next;
          MEMCPY(next,s,w);
          io->next = next;

          m -= w;
  }
  return n-m;
}


Vdbufio_t _Vdbufio =
{ _vdfilbuf,
  _vdflsbuf,
  _vdgetu,
  _vdputu,
  _vdread,
  _vdwrite
};
```

Claims

1. In a computer, the improvement comprising:

   a) program means for producing instructions which allow replication of a second version of a binary file, based on a first version, without access to the second version.

2. A method of replicating a second version of a binary file, which is derived from a first version, comprising the following steps:

   a) creating a sequence of instructions, each of which generates a character sequence, which, when concatenated, replicate the second version; and
   b) executing the instructions.

3. For a digital computer, the improvement comprising:

   a) first program means for

      i) examining a first and a second file and
      ii) producing a sequence of instructions which, when executed, concatenate

         A) parts of the first file with
         B) parts of the second file, without access to the second file,

   in order to replicate the second file.

4. The improvement according to claim 3, and further comprising second program means for executing the sequence of instructions, for replicating the second file.

5. A method of saving a later version of a computer file, comprising the following steps:

   a) comparing the later version with an earlier version, and identifying

      i) similar passages, where the later version is similar to the earlier version, and
      ii) difference passages, where the later version is different from the earlier version;

   b) storing the address where each similar passage occurs in the earlier version; and
   c) storing each difference passage, together with the address of its occurrence in the later version.

6. A process of information replication, comprising the following steps:

   a) finding, at a first site,

      i) similarities and
      ii) differences

   between a first file and a second file;
   b) maintaining, at a second site, a copy of the first file;
   c) transmitting the following to the second site:

      i) locations of the similarities,
      ii) the differences themselves, and
      iii) information to enable replication of the second file, based on (c)(i), (c)(ii), and the second file.

7. A method of updating an old version of a file into a new version, comprising the following steps:

   a) receiving instructions which order one or more of the following to occur:

      i) copying a specified part of the old version into a specified location of the new version;
      ii) copying a specified part of the new version into a specified location of the new version; and
      iii) adding specified bytes to a specified location in the new version;

   and
   b) executing the instructions.

8. A system for generating data indicative of differences between first and second versions of a file, comprising:

   a) means for deriving a sequence comprising the following instructions:

      i) a COPY instruction which orders that characters be copied from

         A) a source location in either the first or second version, to
         B) a destination location in the second version; and

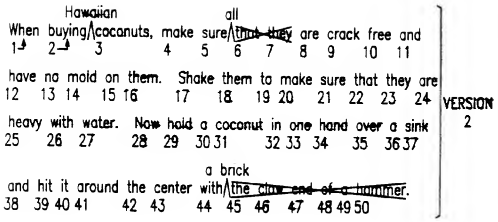
      ii) an ADD instruction which orders that specified characters be added to specified locations in the second version.

```
WORD ——►When  buying  coconuts,  make  sure  that  they  are  crack  free  and
POSITION—►1↲   2↲      3          4     5     6     7     8     9      10    11

        have  no  mold  on  them.  Shake  them  to  make  sure  that  they  are
        12    13  14    15  16     17     18    19  20    21    22    23    24     VERSION
                                                                                     1
        heavy  with  water.  Now  hold  a  coconut  in  one  hand  over  a  sink
        25     26    27      28   29    30 31       32  33   34    35    36 37

        and  hit  it  around  the  center  with  the  claw  end  of  a  hammer.
        38   39   40  41      42   43      44    45   46    47   48 49 50
```

VOCABULARY

1 When	16 them	31 it
2 buying	17 Shake	32 around
3 coconuts	18 to	33 the
4 make	19 heavy	34 center
5 sure	20 with	35 claw
6 that	21 water	36 end
7 they	22 Now	37 of
8 are	23 hold	38 hammer
9 crack	24 a	
10 free	25 in	
11 and	26 one	
12 have	27 hand	
13 no	28 over	
14 mold	29 sink	
15 on	30 hit	

FIG. 1A



```
              Hawaiian                    all
When buying^coconuts, make sure^~~that they~~ are crack free and
1     2      3         4    5     6    7    8    9     10   11

have no mold on them.  Shake them to make sure that they are
12   13  14  15  16    17    18   19  20   21   22   23  24

heavy with water.  Now hold a coconut in one hand over a sink
25    26   27      28  29   30 31     32 33  34   35  36 37

                                      a brick
and hit it around the center with^~~the claw end of a hammer~~.
38  39 40 41      42  43     44   45  46   47  48 49 50
```

VERSION 2

FIG. 1B

When buying $\boxed{\text{Hawaiian}}$ coconuts, make sure $\boxed{\text{all}}$ are crack free and
1 2 3 4 5 6 7 8 9 10 11

have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24

heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37

and hit it around the center with $\boxed{\text{a brick}}$.
38 39 40 41 42 43 44 45 46

VERSION 2

VOCABULARY

1 When	16 them	31 it
2 buying	17 Shake	32 around
3 coconuts	18 to	33 the
4 make	19 heavy	34 center
5 sure	20 with	35 claw
6 that	21 water	36 end
7 they	22 Now	37 of
8 are	23 hold	38 hammer
9 crack	24 a	39 Hawaiian
10 free	25 in	40 all
11 and	26 one	41 brick
12 have	27 hand	
13 no	28 over	
14 mold	29 sink	
15 on	30 hit	

NEW WORDS

FIG. 1C

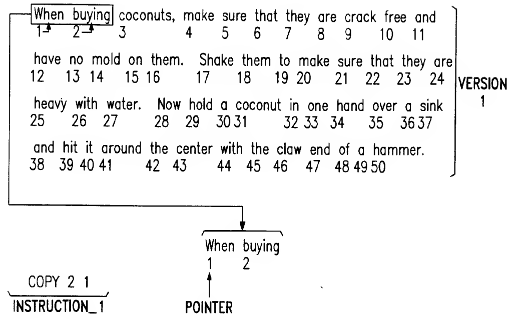

When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
When buying
1 2
COPY 2 1
INSTRUCTION_1
POINTER

FIG. 1D

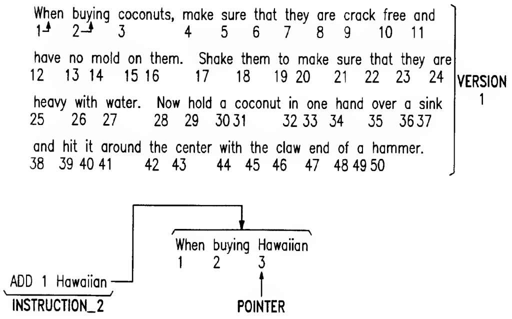
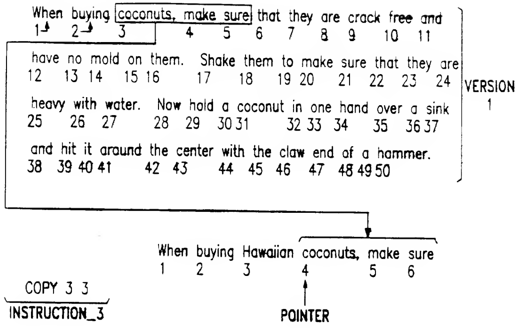
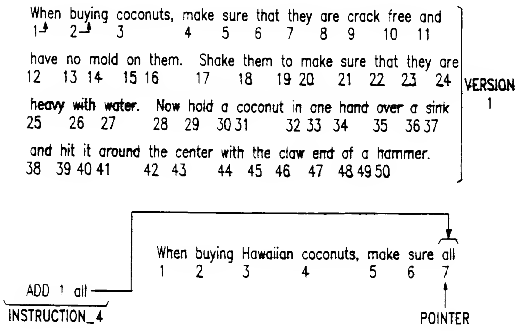

When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
When buying Hawaiian
1 2 3
ADD 1 Hawaiian
INSTRUCTION_2
POINTER

FIG. 1E



When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
When buying Hawaiian coconuts, make sure
1 2 3 4 5 6
COPY 3 3
INSTRUCTION_3
POINTER

FIG. 1F



When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
When buying Hawaiian coconuts, make sure all
1 2 3 4 5 6 7
ADD 1 all
INSTRUCTION_4
POINTER

FIG. 1G

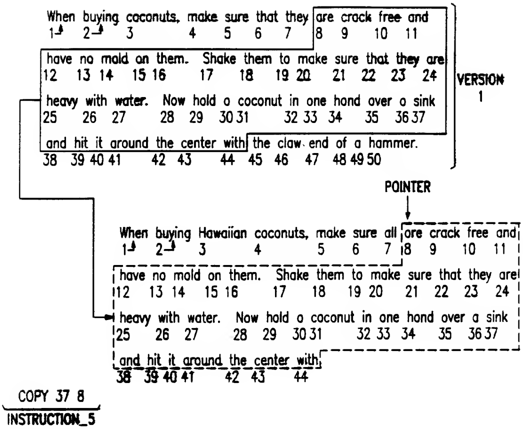

When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
POINTER
When buying Hawaiian coconuts, make sure all are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with
38 39 40 41 42 43 44
COPY 37 8
INSTRUCTION_5

FIG. 1H

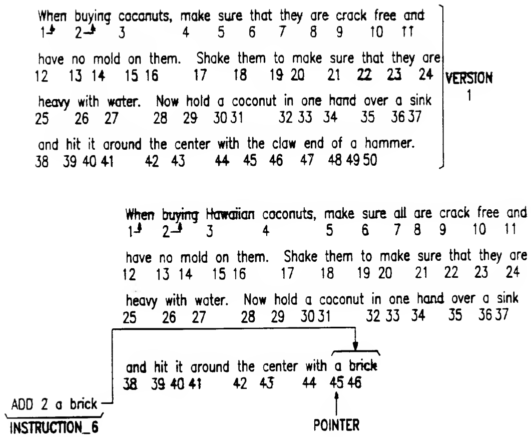

When buying coconuts, make sure that they are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with the claw end of a hammer.
38 39 40 41 42 43 44 45 46 47 48 49 50
VERSION 1
When buying Hawaiian coconuts, make sure all are crack free and
1 2 3 4 5 6 7 8 9 10 11
have no mold on them. Shake them to make sure that they are
12 13 14 15 16 17 18 19 20 21 22 23 24
heavy with water. Now hold a coconut in one hand over a sink
25 26 27 28 29 30 31 32 33 34 35 36 37
and hit it around the center with a brick
38 39 40 41 42 43 44 45 46
ADD 2 a brick
INSTRUCTION_6
POINTER

FIG. 1I

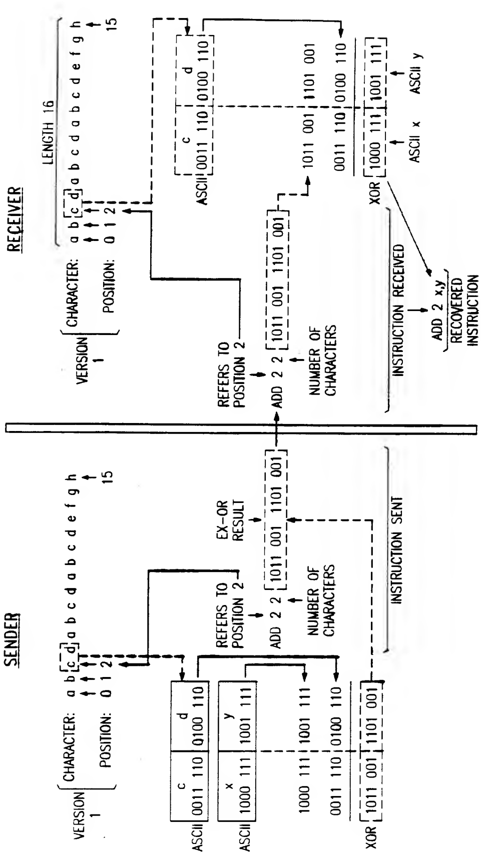

SENDER
VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 15
ASCII c d 0011 110 0100 110
ASCII x y 1000 111 1001 111
1000 111 1001 111
0011 110 0100 110
XOR 1011 001 1101 001
REFERS TO POSITION 2
ADD 2 2 1011 001 1101 001
NUMBER OF CHARACTERS
EX-OR RESULT
INSTRUCTION SENT
RECEIVER
VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 15
LENGTH 16
ASCII c d 0011 110 0100 110
REFERS TO POSITION 2
ADD 2 2 1011 001 1101 001
NUMBER OF CHARACTERS
1011 001 1101 001
0011 110 0100 110
XOR 1000 111 1001 111
ASCII x ASCII y
INSTRUCTION RECEIVED
ADD 2 x,y
RECOVERED INSTRUCTION

FIG. 1J

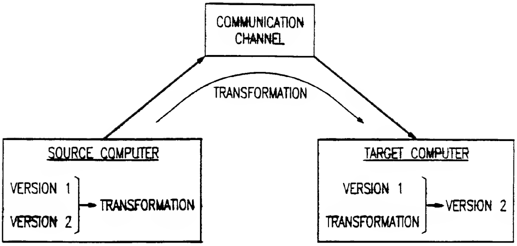

COMMUNICATION CHANNEL
TRANSFORMATION
SOURCE COMPUTER
VERSION 1
VERSION 2
TRANSFORMATION
TARGET COMPUTER
VERSION 1
TRANSFORMATION
VERSION 2

FIG. 1

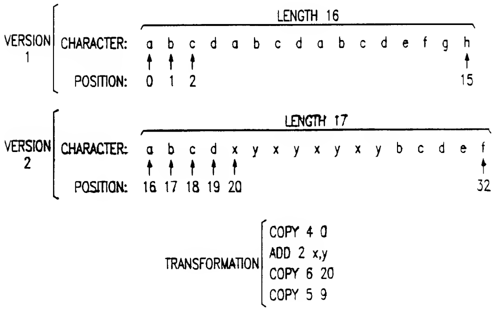

LENGTH 16
VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 15
LENGTH 17
VERSION 2
CHARACTER: a b c d x y x y x y x y b c d e f
POSITION: 16 17 18 19 20 32
TRANSFORMATION
COPY 4 0
ADD 2 x,y
COPY 6 20
COPY 5 9

FIG. 2A

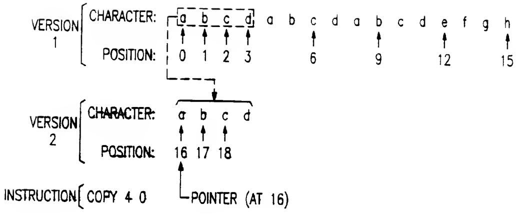

VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 3 6 9 12 15
VERSION 2
CHARACTER: a b c d
POSITION: 16 17 18
INSTRUCTION COPY 4 0
POINTER (AT 16)

FIG. 2B

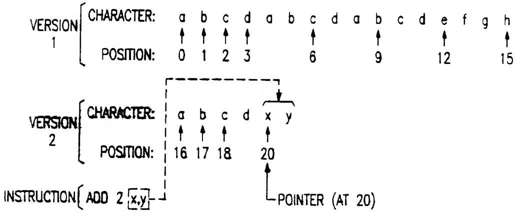
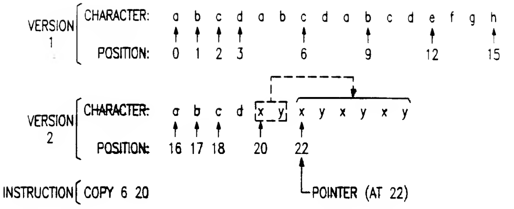
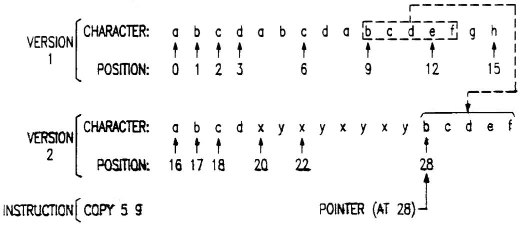

VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 3 6 9 12 15
VERSION 2
CHARACTER: a b c d x y
POSITION: 16 17 18 20
INSTRUCTION ADD 2 x,y
POINTER (AT 20)

FIG. 2C



VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 3 6 9 12 15
VERSION 2
CHARACTER: a b c d x y x y x y x y
POSITION: 16 17 18 20 22
INSTRUCTION COPY 6 20
POINTER (AT 22)

FIG. 2D



VERSION 1
CHARACTER: a b c d a b c d a b c d e f g h
POSITION: 0 1 2 3 6 9 12 15
VERSION 2
CHARACTER: a b c d x y x y x y x y b c d e f
POSITION: 16 17 18 20 22 28
INSTRUCTION COPY 5 9
POINTER (AT 28)

FIG. 2E

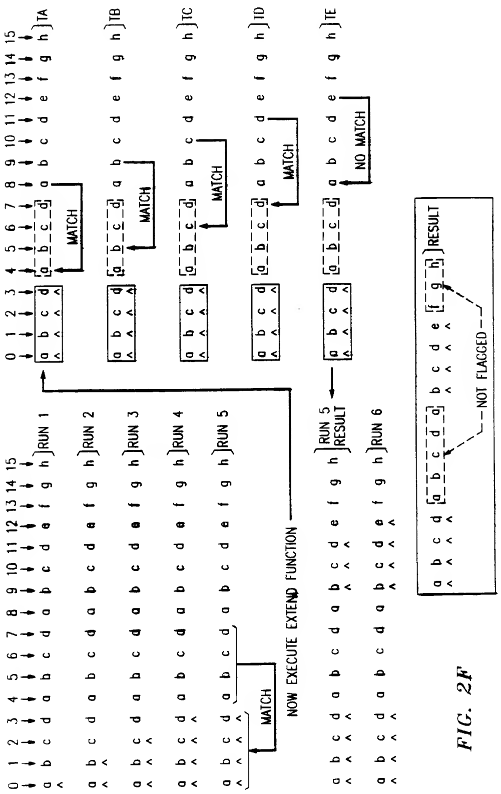
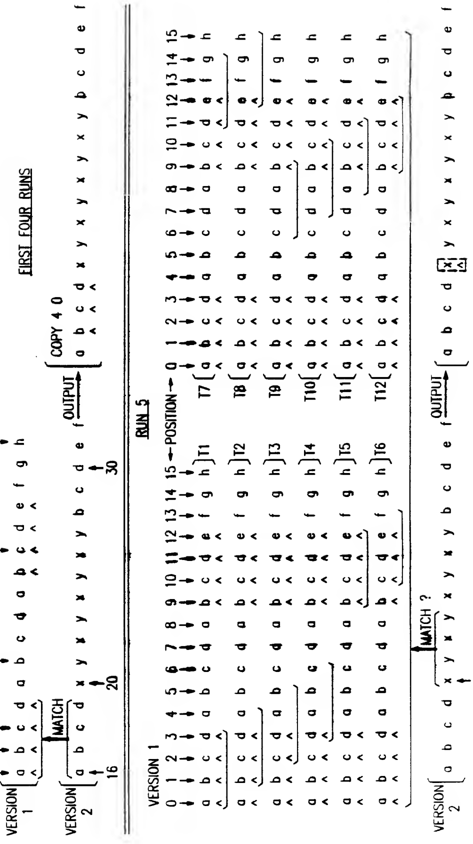

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
MATCH
NOW EXECUTE EXTEND FUNCTION
RUN 5 RESULT
RUN 6
TA
TB
TC
TD
TE
MATCH
NO MATCH
RESULT
NOT FLAGGED

FIG. 2F

FIG. 2G


FIRST FOUR RUNS
VERSION 1
VERSION 2
MATCH
OUTPUT
COPY 4 0
RUN 5
VERSION 1
POSITION
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
MATCH ?
VERSION 2
OUTPUT

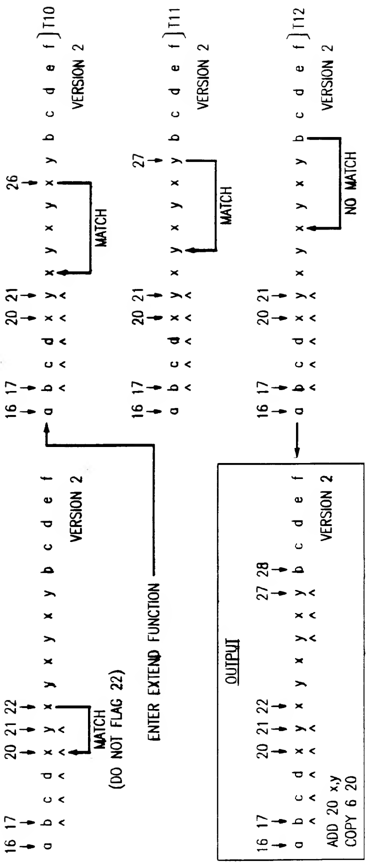

16 17 20 21 22
a b c d x y x y x y x y b c d e f
VERSION 2
MATCH
(DO NOT FLAG 22)
ENTER EXTEND FUNCTION
OUTPUT
16 17 20 21 22 27 28
ADD 20 x,y
COPY 6 20
16 17 20 21 26
MATCH
T10
27
T11
NO MATCH
T12

FIG. 2H

```
VERSION1 [ a b c d a b c d a b c d e f g h

VERSION2 [ a b c d x y x y x y x y b c d e f

               [ 1. COPY  4  0
               [ 2. ADD   2  x,y
TRANSFORMATION [ 3. COPY  6  20
               [ 4. COPY  5  9
```

FIG. 2

```
1.  n = length(VERSION1);
2.  C = 0
3.  while(true)
4.  {   inst = readinst();
5.      if(inst == end-of-file)
6.           return;
7.      s = readsize();
8.      if(inst == ADD)
9.           readdata(VERSION2+c,s);
10.     else
11.     {    p = readpos();
12.          if(p < n)
13.               copy(VERSION2+c,VERSION1+p,s);
14.          else copy(VERSION2+c,VERSION2+p-n,s);
15.     }
16.     c = c + s;
17. }
```

FIG. 3

```
01000100    0
00000010    x y
01000110    20
01000101    9
```

FIG. 4

```
Initialize table of positions T to empty;
process(VERSION1);
process(VERSION2);
process(seq)
{   n = length(VERSION1);
    m = length(seq);
    c = 0;
    add = -1;
    pos = -1;
    len = MIN-1;
    while(1)
    {   while(1)
        {   p = search(T,seq,c,len);
            if(p < 0)
                break;
            len = extend(T,seq,c,p,len);
            pos = p;
        }
        if(pos < 0)
        {   if(add < 0)
                add = c;
            if(seq == VERSION1)
                insert(T,seq,c,0);
            else insert(T,seq,c,n);
            c = c + 1;
        }
        else
        {   if(seq == VERSION2)
                writeinst(add,c,pos,len);
            p = c+len-(MIN-1);
            while(p < c+len)
            {   if(seq == VERSION1)
                    insert(T,seq,p,0);
                else insert(T,seq,p,n);
                p = p+1
            }
            c = c + len;
            add = -1; pos = -1; len = MIN-1;
        }
        if(c >= m-MIN)
            break;
    }
    if(seq == VERSION2 and add >= 0)
        writeinst(add,m,-1,0);
}
```

FIG. 5

```
insert(T,seq,p,orig)
{   key = seq[p,p+1,...,p+MIN-1];
    pos = p+orig;
    Insert (key,pos) into T;
}
```

FIG. 6

```
search(T,seq,c,len)
{   n = length(VERSION1);
    p = c + len - (MIN-1);
    key = seq[p,p+1,...,p+MIN-1];
    for(each entry e in T that matches key)
    {   pos = position(e);
        if(p >= n)
        {   str = VERSION2;
            q = pos-n;
        }
        else
        {   str = VERSION1;
            q = pos
        }
        d = q - (len - (MIN-1) );
        if(d >= 0)
            if(seq[c,c+1,...,p-1] == str[d,d+1,...,q-1])
                return pos - (len - (MIN-1) );
    }
    return -1;
}
```

FIG. 7

```
extend(T,seq,c,p,len)
{   n = length(VERSION1);
    if(p < n)
        str = VERSION1;
    else
    {   str = VERSION2;
        p = p-n;
    }
    m = length(seq);  i = c + len + 1;
    n = length(str);  j = p + len + 1;
    while(i < m and j < n)
        if(seq[i] != seq[j])
            break;
    return i-c;
}
```

FIG. 8

```
VERSION1 [ a b c d a b c d a b c d e f g h
           ^ ^ ^ ^           ^ ^ ^ ^

VERSION2 [ a b c d x y x y x y x y b c d e f
             ^ ^ ^ ^ ^       ^ ^ ^
```

FIG. 9

```
01000100    0
00000010    2     x^c  y^d
01000110    20
01000101    9
```

FIG. 10